2.1 Numbers

Much of the data processed by computers consists of numbers. In programming terminology, numbers are called **numeric literals**. This section discusses the operations that are performed with numbers and the ways numbers are displayed.

Two Types of Numbers: *ints* and *floats*

A whole number written without a decimal point is called an **int** (short for *integer*) and a number written with a decimal point is called a **float** (short for *floating-point* number).

NUMBER	TYPE	NUMBER	TYPE
34	int	23.45	float
34.	float	−34	int

Arithmetic Operators

The five basic arithmetic operations are addition, subtraction, multiplication, division, and exponentiation. The addition, subtraction, and division operators are denoted in Python by the standard symbols +, −, and /, respectively. However, the notations for the multiplication and exponentiation operators differ from the customary mathematical notations.

MATHEMATICAL NOTATION	MEANING	PYTHON NOTATION
$a \cdot b$ or $a \times b$	a times b	a * b
a^r	a to the r^{th} power	a ** r

The result of a division is always a **float**, even if the quotient evaluates to a whole number. The result of the other operations is a **float** if either of the numbers is a **float** and otherwise is an **int**.

The *print* Function

The **print function** is used to display numbers on the monitor. If n is a number, then the statement

```
print(n)
```

displays the number n. A combination of numbers, arithmetic operators, and parentheses that can be evaluated is called a **numeric expression**. The **print** function applied to an expression displays the result of evaluating the expression. A single **print** function can display several values. If m, n, r, . . . are numbers (or numeric expressions), then the statement

```
print(m, n, r, . . .)
```

displays the numbers (or values of the numeric expressions) one after another separated by spaces.

The **print** function invokes a **newline operation** that causes the next **print** function to display its output at the beginning of a new line.

Example 1 Arithmetic Operations The following program applies each of the five standard arithmetic operations. [Run] indicates that the program should be executed (by pressing the F5 key or clicking on *Run Module* in the *Run* menu). The lines after [Run] show

the output of the program. In the evaluation of 2 * (3 + 4), the operation inside the parentheses was calculated first. (Expressions inside parentheses are *always* evaluated first.)

```
print(3 + 2, 3 - 2, 3 * 2)
print(8 / 2, 8 ** 2, 2 * (3 + 4))
```

[Run]

```
5 1 6
4.0 64 24
```

Note: All programs appearing in examples can be downloaded from the companion website for this book. See the discussion in the preface for details.

Variables

In mathematics problems, quantities are referred to by names. For instance, consider the following algebra problem: "If a car travels at 50 miles per hour, how far will it travel in 14 hours?" The solution to this problem uses the well-known formula

$$\text{distance} = \text{speed} \times \text{time elapsed}.$$

Example 2 shows how this problem would be solved with a Python program.

Example 2 **Distance Traveled** The following program uses the speed and the time elapsed to calculate the distance traveled. The names given to the values are called **variables**. The first line of the program is said to create (or declare) the variable *speed* and to assign it the value 50. Similarly, the second and third lines create and assign values to other variables.

```
speed = 50
timeElapsed = 14
distance = speed * timeElapsed
print(distance)
```

[Run]

```
700
```

Numeric expressions may also contain variables. Expressions are evaluated by replacing each variable by its value and then carrying out the arithmetic. Some examples of expressions containing variables are (2 * distance) + 7, $n + 1$, and $(a + b)/3$.

In general, a variable is a name that refers to an item of data stored in memory. In this section of the book, all data will be numbers. A statement of the form


VideoNote
Assignment Statements

$$variableName = numericExpression$$

is called an **assignment statement**. The statement first evaluates the expression on the right and then assigns its value to the variable on the left. The variable is created the first time it appears on the left side of an assignment statement. Subsequent assignment statements for the variable alter the value assigned to the variable. Actually, each variable points to a location in memory that stores the value. A variable must first be created with an assignment statement before it can be used in an expression.

In Python, variable names must begin with a letter or an underscore, and can consist only of letters, digits, and underscores. (The shortest variable names consist of a single

letter.) Descriptive variable names help others (and you at a later time) easily recall what the variable represents. Some examples of descriptive variable names are *totalSales*, *rateOfChange*, and *taxRate*. As a convention, we write variable names in lowercase letters except for the first letters of each additional word. This naming convention is called **camel casing** since the uppercase letters appear to create humps in the name.

Python is case-sensitive, that is, it distinguishes between uppercase and lowercase letters. Therefore, the variables *amount* and *Amount* are different variables.

There are 33 words, called **reserved words** (or **keywords**), that have special meanings in Python and cannot be used as variable names. Some examples of reserved words are *return*, *for*, *while*, and *def*. Appendix B lists the 33 reserved words. (***Note:*** IDLE automatically color codes reserved words in the color orange.)

The *abs*, *int*, and *round* Functions

There are several common operations that can be performed on numbers other than the standard arithmetic operations. For instance, we may round a number or take its absolute value. These operations are performed by built-in functions. Functions associate with one or more values, called the *input*, a single value called the *output*. The function is said to **return** the output value. The three functions considered in the next paragraph have numeric input and output.

The absolute value function, `abs(x)`, is `|x|`. The function strips the minus signs from negative numbers while leaving other numbers unchanged. The `int` function leaves integers unchanged, and converts floating-point numbers to integers by discarding their decimal part. The value of `round(n, r)` is the number *n* rounded to *r* decimal places. The argument *r* can be omitted. If so, *n* is rounded to a whole number. Some examples are as follows:

EXPRESSION	VALUE	EXPRESSION	VALUE	EXPRESSION	VALUE
abs(3)	3	int(2.7)	2	round(2.7)	3
abs(0)	0	int(3)	3	round(2.317, 2)	2.32
abs(−3)	3	int(−2.7)	−2	round(2.317, 1)	2.3

The terms inside the parentheses can be numbers (as shown), numeric variables, or numeric expressions. Expressions are evaluated to produce the input.

Example 3 **Functions** The following program evaluates each of the preceding three functions at an expression:

```
a = 2
b = 3
print(abs(1 - (4 * b)))
print(int((a ** b) + .8))
print(round(a / b, 3))
```

[Run]

```
11
8
0.667
```

Note: Function names, like variable names, are case-sensitive. For instance, the `round` function cannot be written *Round*.

Augmented Assignments

Since the expression on the right side of an assignment statement is evaluated *before* the assignment is made, a statement such as

```
var = var + 1
```

is meaningful. It first evaluates the expression on the right (that is, it adds 1 to the value of the variable *var*) and then assigns this sum to the variable *var*. The effect is to increase the value of the variable *var* by 1. In terms of memory locations, the statement retrieves the value of *var* from *var*'s memory location, uses it to compute *var* + 1, and then places the sum into a memory location. This type of calculation is so common that Python provides a special operator to carry it out. The statement

```
var = var + 1
```

can be replaced with the statement

```
var += 1
```

In general, if *n* has a numeric value, then the statement

```
var += n
```

adds the value of *n* to the value of *var*. The operator += is said to perform an **augmented assignment**. Some other augmented assignment operators are −=, *=, /=, and **=.

Example 4 Augmented Assignments The following program illustrates the different augmented assignment operators.

```
num1 = 6
num1 += 1
num2 = 7
num2 -= 5
num3 = 8
num3 /= 2
print(num1, num2, round(num3))
num1 = 1
num1 *= 3
num2 = 2
num2 **= 3
print(num1, num2)
```

[Run]

```
7 2 4
3 8
```

Two Other Integer Operators

In addition to the five standard arithmetic operators discussed at the beginning of this section, the **integer division operator** (written //) and the **modulus operator** (written %) are also available in Python. Let *m* and *n* be positive whole numbers. When you use long division to divide *m* by *n*, you obtain an integer quotient and an integer remainder. In

Python, the integer quotient is denoted m // n, and the integer remainder is denoted m % n. For instance,

$$\begin{array}{r} 4 \leftarrow 14 \text{ // } 3 \\ 3\overline{)\,14} \\ \underline{12} \\ 2 \leftarrow 14 \text{ \% } 3 \end{array}$$

Essentially, m // n divides two numbers and chops off the fraction part, and m % n is the remainder when m is divided by n. Some examples are as follows:

EXPRESSION	VALUE	EXPRESSION	VALUE
19 // 5	3	19 % 5	4
10 // 2	5	10 % 2	0
5 // 7	0	5 % 7	5

Example 5 **Convert Lengths** The following program converts 41 inches to 3 feet and 5 inches:

```
totalInches = 41
feet = totalInches // 12
inches = totalInches % 12
print(feet, inches)
```

[Run]

```
3 5
```

Parentheses, Order of Precedence

Parentheses should be used to clarify the meaning of an expression. When there are insufficient parentheses, the arithmetic operations are performed in the following order of precedence:

1. terms inside parentheses (inner to outer)
2. exponentiation
3. multiplication, division (ordinary and integer), modulus
4. addition and subtraction.

In the event of a tie, the leftmost operation is performed first. For instance, 8 / 2 * 3 is evaluated as (8 / 2) * 3.

A good programming practice is to use parentheses liberally so that you *never* have to remember the order of precedence. For instance, write (2 * 3) + 4 instead of 2 * 3 + 4 and write 4 + (2 ** 3) instead of 4 + 2 ** 3.

Three Kinds of Errors

Grammatical and punctuation errors are called **syntax errors**. Some incorrect statements and their errors are shown in Table 2.1.

TABLE 2.1 Three syntax errors.

Statement	Reason for Error
`print(3))`	The statement contains an extraneous right parenthesis.
`for = 5`	A reserved word is used as a variable name.
`print(2; 3)`	The semicolon should be a comma.

If a syntax error is spotted when the code is analyzed by the interpreter (that is, before the program begins to execute), Python displays a message box similar to one of those in Fig. 2.1. After you click on the *OK* button, Python will display the program with a blinking cursor placed near the location of the error.

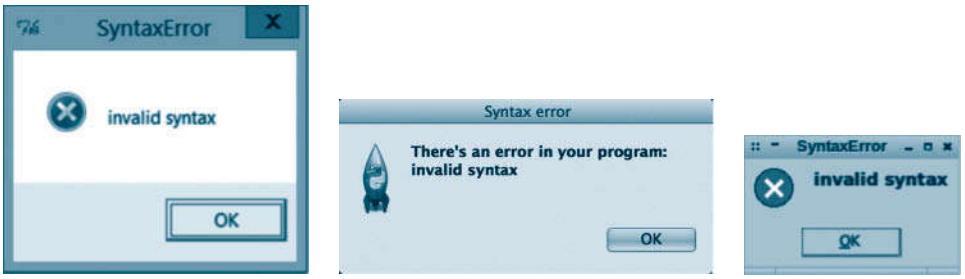

FIGURE 2.1 Syntax error message boxes.

Errors that are discovered while a program is running are called **runtime errors** or **exceptions**. Some incorrect statements and their errors are shown in Table 2.2.

TABLE 2.2 Three runtime errors.

Statement	Reason for Error
`primt(5)`	The function `print` is misspelled.
`x += 1`, when *x* has not been created	Python is not aware of the variable *x*.
`print(5 / 0)`	A number cannot be divided by zero.

The first two errors in Table 2.2 are said to be of the type *NameError*, and the third is said to be of the type *ZeroDivisionError*. When Python encounters an exception, Python terminates execution of the program and displays a message such as the one in Fig. 2.2. The last two lines of the error message identify the statement that caused the error and give its type.

```
Traceback (most recent call last):
  File "C:\test1.py", line 2, in <module>
    print(5 / 0) #ZeroDivisionError
ZeroDivisionError: division by zero
```

FIGURE 2.2 An error message for an exception error.

A third kind of error is called a **logic error**. Such an error occurs when a program does not perform the way it was intended. For instance, the statement

```
average = firstNum + secondNum / 2
```

is syntactically correct. However, an incorrect value will be generated, since the correct way to calculate an average is

```
average = (firstNum + second Num) / 2
```

Logic errors are the most difficult kind of error to locate.

Numeric Objects in Memory

Consider the following lines of code:

```
n = 5
n = 7
```

Figure 2.3 shows what happens in memory when the two lines of code are executed. When the first line of code is executed, Python sets aside a portion of memory to hold the number 5. The variable *n* is said to **reference** (or **point to**) the number 5 in the memory location. When the second line of code is executed, Python sets aside a new memory location to hold the number 7 and redirects the variable *n* to point to the new memory location. The number 5 in memory is said to be *orphaned* or *abandoned*. Python will eventually remove the orphaned number from memory with a process called *garbage collection*.

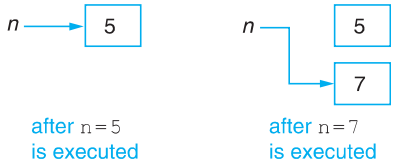

FIGURE 2.3 Numeric objects in memory.

Comments

1. Names given to variables are sometimes referred to as *identifiers*.
2. A numeric expression is any combination of literals, variables, functions, and operators that can be evaluated to produce a number. A single literal or variable is a special case of an expression.
3. Numeric literals used in expressions or assigned to variables must not contain commas, dollar signs, or percent signs. Also, mixed numbers, such as 8 1/2, are not allowed.
4. When the number *n* is halfway between two successive whole numbers (such as 1.5, 2.5, 3.5, and 4.5), the round function rounds it to the nearest even number. For instance, round(2.5) is 2 and round(3.5) is 4.
5. In scientific notation, numbers are written in the form $b \cdot 10^r$, where b is a number of magnitude from 1 up to (but not including) 10, and r is an integer. Python often displays very large and very small numbers in **scientific notation**, where $b \cdot 10^r$ is written as ***be+r*** or ***be-r***. (The letter *e* is an abbreviation for *exponent*.) For instance, when the statement `print(123.8 * (10 ** 25))` is executed, 1.238e+27 is displayed.
6. The functions discussed in this section are referred to as **built-in functions** since they are part of the Python language. Chapter 4 shows how we can create our own functions. Such functions are commonly referred to as *user-defined functions*. The term

user-defined is a bit of a misnomer; such functions should really be called *programmer-defined functions*.

8. IDLE color codes the different types of elements. For instance, normal text is displayed in black and built-in functions (such as **print**, **abs**, **int**, and **round**) are displayed in purple.
9. The word "exception" is shorthand for "exceptional (that is, bad) event."

Practice Problems 2.1

1. Evaluate $7 - 4 \% 3$.
2. Explain the difference between the assignment statement

```
var1 = var2
```

and the assignment statement

```
var2 = var1
```

3. Complete the table by filling in the value of each variable after each line of code is executed.

	a	b	c
`a = 5`	5	does not exist	does not exist
`b = 4`	5	4	does not exist
`c = a * b`	5	4	20
`a = c // a`			
`print((a - b) * c)`			
`b = b * b * b`			

4. Write a statement that increases the value of the numeric variable *var* by 5%.

EXERCISES 2.1

In Exercises 1 through 12, evaluate the numeric expression without the computer, and then use Python to check your answer.

1. 1.7 * 8
2. 7 ** 2
3. 1 / (2 ** 3)
4. 3 + (4 * 5)
5. (8 + 6) / 5
6. 3 * ((−2) ** 5)
7. 7 // 3
8. 14 % 4
9. 9 % 3
10. 14 // 4
11. 2 ** 2
12. 5 % 5

In Exercises 13 through 18, determine whether the name is a valid variable name.

13. _salesman1
14. room&Board
15. fOrM_1040
16. 1040B
17. @variable
18. INCOME 2008

In Exercises 19 through 24, evaluate the numeric expression where $a = 5$, $b = 3$, and $c = 7$.

19. `(a * b) + c`
20. `a * (b + c)`
21. `(1 + b) * c`
22. `a ** c`
23. `b ** (c - a)`
24. `(c - a) ** b`

In Exercises 25 through 30, write lines of code to calculate and display the values.

25. $5 \cdot 3 + 3 \cdot 5$

26. $(3^4) \cdot (4^3)$

27. 200 + 10% of 100

28. $(2^3 - 1) + 5$

29. $31 \cdot (2 + 28)$

30. $\frac{1}{2^4} \cdot 2^4$

In Exercises 31 and 32, complete the table by filling in the value of each variable after each line is executed.

31.

	x	y
`x = -2`		
`y = x + 5`		
`x = x ** y`		
`print((x/y) + 2)`		
`y = y % 2 + 0.6`		

32.

	bal	inter	withDr
`bal = 100`			
`inter = .05`			
`withDr = 25`			
`bal += (inter * bal)`			
`bal = bal - withDr`			

In Exercises 33 through 38, determine the output displayed by the lines of code.

33.
```
a = 3
b = 5
print(a * b ** 2)
```

34.
```
d = 5
d -= 1
print(d, d + 1, d - 2)
```

35.
```
n = 5
n ** = 2
print(n/5)
```

36.
```
points = 30
points += 20 * 10
print(points)
```

37.
```
totalBerries = 100
totalCost = 352
eachBerry = totalCost /
            totalBerries
print(eachBerry)
```

38.
```
totalMeters = 30255
kiloMeters = totalMeters // 1000
meters = totalMeters % 1000
print(kiloMeters, meters)
```

In Exercises 39 through 42, identify the errors.

39.
```
a = 2
b = 3
a + b = c
print(b)
```

40.
```
balance = 1,234
deposit = $100
print(Balance + Deposit)
```

41.
```
0.05 = interest
balance = 800
print(interest * balance)
```

42.
```
9W = 2 * 9W
print(9W)
```

In Exercises 43 through 48, find the value of the function.

43. `int(10.75)`

44. `int(9 - 2)`

45. `abs(3 - 10)`

46. `abs(10 ** (-3))`

47. `round(3.1279, 3)`

48. `round(-2.6)`

In Exercises 49 through 54, find the value of the function where $a = 6$ and $b = 4$.

49. `int(-a / 2)`

50. `round(a / b)`

51. `abs(a - 5)`

52. `abs(4 - a)`

53. `round(a + 0.5)`

54. `int(b * 0.5)`

In Exercises 55 through 60, rewrite the statements using augmented assignment operators.

55. `cost = cost + 5`

56. `sum = sum * 2`

57. `cost = cost / 6`

58. `sum = sum - 7`

59. `sum = sum % 2`

60. `cost = cost // 3`

In Exercises 61 through 68, write a program that has one line of code for each step.

61. Calculate Profit The following steps calculate a company's profit.

- **(a)** Create the variable *revenue* and assign it the value 98,456.
- **(b)** Create the variable *costs* and assign it the value 45,000.
- **(c)** Create the variable *profit* and assign it the difference between the values of the variables *revenue* and *costs*.
- **(d)** Display the value of the variable *profit*.

62. Stock Purchase The following steps calculate the amount of a stock purchase.

- **(a)** Create the variable *costPerShare* and assign it the value 25.625.
- **(b)** Create the variable *numberOfShares* and assign it the value 400.
- **(c)** Create the variable *amount* and assign it the product of the values of *costPerShare* and *numberOfShares*.
- **(d)** Display the value of the variable *amount*.

63. Discounted Price The following steps calculate the price of an item after a 30% reduction.

- **(a)** Create the variable *price* and assign it the value 19.95.
- **(b)** Create the variable *discountPercent* and assign it the value 30.
- **(c)** Create the variable *markdown* and assign it the value of (*discountPercent* divided by 100) times the value of *price*.
- **(d)** Decrease the value of *price* by *markdown*.
- **(e)** Display the value of *price* (rounded to two decimal places).

64. Break-Even Point The following steps calculate a company's break-even point, the number of units of goods the company must manufacture and sell in order to break even.

- **(a)** Create the variable *fixedCosts* and assign it the value 5,000.
- **(b)** Create the variable *pricePerUnit* and assign it the value 8.
- **(c)** Create the variable *costPerUnit* and assign it the value 6.
- **(d)** Create the variable *breakEvenPoint* and assign it the value of *fixedCosts* divided by (the difference of the values of *pricePerUnit* and *costPerUnit*).
- **(e)** Display the value of the variable *breakEvenPoint*.

65. Savings Account The following steps calculate the balance after three years when $100 is deposited in a savings account at 5% interest compounded annually.

- **(a)** Create the variable *balance* and assign it the value 100.
- **(b)** Increase the value of the variable *balance* by 5%.
- **(c)** Increase the value of the variable *balance* by 5%.

(d) Increase the value of the variable *balance* by 5%.
(e) Display the value of *balance* (rounded to two decimal places).

66. Savings Account The following steps calculate the balance at the end of three years when $100 is deposited at the beginning of each year in a savings account at 5% interest compounded annually.

(a) Create the variable *balance* and assign it the value 100.
(b) Increase the value of the variable *balance* by 5%, and add 100 to it.
(c) Increase the value of the variable *balance* by 5%, and add 100 to it.
(d) Increase the value of the variable *balance* by 5%.
(e) Display the value of *balance* (rounded to two decimal places).

67. Savings Account The following steps calculate the balance after 10 years when $100 is deposited in a savings account at 5% interest compounded annually.

(a) Create the variable *balance* and assign it the value 100.
(b) Multiply the value of the variable *balance* by 1.05 raised to the 10^{th} power.
(c) Display the value of *balance* (rounded to two decimal places).

68. Profit from Stock The following steps calculate the percentage profit from the sale of a stock.

(a) Create the variable *purchasePrice* and assign it the value 10.
(b) Create the variable *sellingPrice* and assign it the value 15.
(c) Create the variable *percentProfit* and assign it 100 times the value of the difference between *sellingPrice* and *purchasePrice* divided by *purchasePrice*.
(d) Display the value of the variable *percentProfit*.

In Exercises 69 through 78, write a program to solve the problem and display the answer. The program should use variables for each of the quantities.

69. Corn Production Suppose each acre of farmland produces 18 tons of corn. How many tons of corn can be grown on a 30-acre farm?

70. Projectile Motion Suppose a ball is thrown straight up in the air with an initial velocity of 50 feet per second and an initial height of 5 feet. How high will the ball be after 3 seconds? ***Note:*** The height after t seconds is given by the expression $-16t^2 + v_0t + h_0$, where v_0 is the initial velocity and h_0 is the initial height.

71. Distance Covered If a car left the airport at 5 o'clock and arrived home at 9 o'clock, what was the distance covered? ***Note:*** Speed of the car is 81.34 km per hour.

72. Gas Mileage A motorist wants to determine her gas mileage. At 23,352 miles (on the odometer) the tank is filled. At 23,695 miles the tank is filled again with 14 gallons. How many miles per gallon did the car average between the two fillings?

73. Power Usage A survey showed that the average monthly electricity consumption for a city was 750 million watts per month. What was the daily power consumption in watts of each resident? ***Note:*** The city has a population of about 5 million people.

74. Square Deck José is building a square deck at the back of his house. José has a building permit for a 432-square-foot deck. How long will each side of the deck be?

75. Banks A bank offers 8.7% interest per year on all savings accounts. If a savings account initially contains $1000, how much money will the account hold two years later?

76. Population Increase You grew up in a tiny village and had to move to a nearby city for your undergraduate. When you left, the population was 845. You recently heard that the population of your village has grown by 6.5%. What is the present population of the village? Round the population to the nearest whole number.

77. Bacterial Growth Suppose a surface initially contained $2.19 \cdot 10^{14}$ bacterial cells. After some time, the surface contained $4.68 \cdot 10^{14}$ bacterial cells. Calculate the percentage of bacterial growth. Display the answer rounded to the nearest whole number.

78. Calories Estimate the number of calories in one cubic mile of chocolate ice cream. ***Note:*** There are 5,280 feet in a mile and one cubic foot of chocolate ice cream contains about 48,600 calories.

Solutions to Practice Problems 2.1

1. 6. Modulus operations are performed before subtractions. If the intent is for the subtraction to be performed first, the expression should be written (7 − 4) % 3.
2. The first assignment statement assigns the value of the variable *var2* to the variable *var1*, whereas the second assignment statement assigns *var1*'s value to *var2*.
3.

	a	b	c
`a = 5`	5	does not exist	does not exist
`b = 4`	5	4	does not exist
`c = a * b`	5	4	20
`a = c // a`	4	4	20
`print((a - b)*c)`	4	4	20
`b = b * b * b`	4	64	20

Each time an assignment statement is executed, only one variable (the variable to the left of the equal sign) has its value changed.

4. Each of the following four statements increases the value of *var* by 5%.

```
var = var + (.05 * var)
var = 1.05 * var
var += .05 * var
var *= 1.05
```

2.2 Strings

The most common types of data processed by Python are strings and numbers. Sentences, phrases, words, letters of the alphabet, names, telephone numbers, addresses, and social security numbers are all examples of strings.

String Literals

A **string literal** is a sequence of characters that is treated as a single item. The characters in strings can be any characters found on the keyboard (such as letters, digits, punctuation marks, and spaces) and many other special characters.

In Python programs, string literals are written as a sequence of characters surrounded by either single quotes (') or double quotes ("). Some examples of strings are as follows:

"John Doe"
'5th Avenue'

'76'
"Say it ain't so, Joe!"

Opening and closing quotation marks must be the same type—either both double quotes or both single quotes. When a string is surrounded by double quotes, a single quote can appear directly in the string, but not a double quote. Similarly, a string surrounded by single quotes can contain a double quote, but not a single quote directly.[1]

Variables

Variables also can be assigned string values. As with variables assigned numeric values, variables assigned string values are created (that is, come into existence) the first time they appear in assignment statements. When an argument of a **print** function is a string literal or a variable having a string value, only the characters within the enclosing quotation marks (and not the quotation marks themselves) are displayed.

Indices and Slices

In Python, the **position** or **index** of a character in a string is identified with one of the numbers 0, 1, 2, 3, ... For instance, the first character of a string is said to have index 0, the second character is said to have index 1, and so on. If *str1* is a string variable or literal, then str1[*i*] is the character of the string having index *i*. Figure 2.4 shows the indices of the characters of the string "spam & eggs".

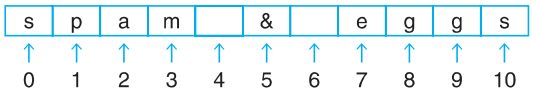

FIGURE 2.4 **Indices of the characters of the string "spam & eggs".**

A **substring** or **slice** of a string is a sequence of consecutive characters from the string. For instance, consider the string "Just a moment". The substrings "Jus", "mom", and "nt" begin at positions 0, 7, and 11, and end at positions 2, 9, and 12, respectively. If *str1* is a string, then `str1[m:n]` is the substring beginning at position *m* and ending at position $n - 1$. Figure 2.5 helps to visualize slices. Think of the indices of the characters pointing just to the left of the characters. Then "spam & eggs"[m:n] is the sequence of characters between the arrows labeled with the numbers *m* and *n*. For instance `"spam & eggs"[2:6]` is the substring `"am &"`; that is, the substring between the arrow labeled 2 and the arrow labeled 6.

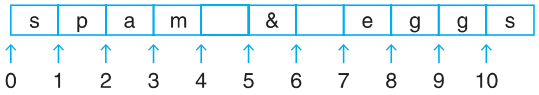

FIGURE 2.5 **Aid to visualizing slices.**

Note: If $m \geq n$, that is, if the character in position *m* is not to the left of the character in position *n*, then the value of `str1[m:n]` will be the **empty string ("")**, the string with no characters.

If *subStr* is a string, then `str1.find(subStr)` is the positive index of the first appearance of *subStr* in *str1* with the search beginning at the left side of the string. The value of `str1.rfind(subStr)` is the positive index of the first appearance of *subStr* in *str1* with the search beginning at the right side of the string. If *subStr* does not appear in *str1*, then the value returned by the **find** and **rfind** methods will be −1.

[1]In Section 2.3, we show how to use escape sequences to override this restriction.

Example 1 Indices The following program illustrates the use of indices.

```
print("Python")
print("Python"[1], "Python"[5], "Python"[2:4])
str1 = "Hello World!"
print(str1.find('W'))
print(str1.find('x'))
print(str1.rfind('l'))
```

[Run]

```
Python
y n th
6
-1
9
```

Negative Indices

The indices discussed above specify positions from the left side of the string. Python also allows strings to be indexed by their position with regards to the right side of the string by using negative numbers for indices. With negative indexing, the rightmost character is assigned index −1, the character to its left is assigned index −2, and so on. Figure 2.6 shows the negative indices of the characters of the string "spam & eggs".

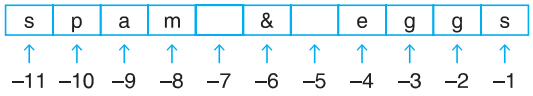

FIGURE 2.6 Negative indices of the characters of the string "spam & eggs".

Example 2 Negative Indices The following program illustrates negative indices.

```
print("Python")
print("Python"[-1], "Python"[-4], "Python"[-5:-2])
str1 = "spam & eggs"
print(str1[-2])
print(str1[-8:-3])
print(str1[0:-1])
```

[Run]

```
Python
n t yth
g
m & e
spam & egg
```

Default Bounds for Slices

In the expression `str1[m:n]`, one or both of the bounds can be omitted. If so, the left bound *m* defaults to 0 and the right bound *n* defaults to the length of the string. That is, `str1[:n]` consists of all the characters from the beginning of the string to `str1[n-1]`, and

`str1[m:]` consists of all the characters from `str1[m]` to the end of the string. The slice `str1[:]` is the entire string *str1*.

Example 3 Default Bounds The following program illustrates default bounds.

```
print("Python"[2:], "Python"[:4], "Python"[:])
print("Python"[-3:], "Python"[:-3])
```

[Run]

```
thon Pyth Python
hon Pyt
```

String Concatenation

Two strings can be combined to form a new string consisting of the strings joined together. This operation is called **concatenation** and is represented by a plus sign. For instance, "good" + "bye" is "goodbye". A combination of strings, plus signs, functions, and methods that can be evaluated to form a string is called a **string expression**. When a string expression appears in an assignment statement or a print function, the string expression is evaluated before being assigned or displayed.

String Repetition

The asterisk operator can be used with strings to repeatedly concatenate a string with itself. If *str1* is a string literal, variable, or expression and *n* is a positive integer, then the value of

```
str1 * n
```

is the concatenation of *n* copies of the value of *str1*.

EXPRESSION	VALUE	EXPRESSION	VALUE
"ha" * 4	"hahahaha"	'x' * 10	"xxxxxxxxxx"
"mur" * 2	"murmur"	("cha-" * 2) + "cha"	"cha-cha-cha"


VideoNote
String Functions

String Functions and Methods

A string function operates much like a numeric function; it takes a string as input and returns a value. A string method is a process that performs a task on a string. We have already seen two examples of methods—the find and rfind methods. These methods perform the task of locating an index. The general form of an expression applying a method is

```
stringName.methodName()
```

where the parentheses might contain values. Like the numeric functions discussed in the previous section, string functions and methods also can be applied to literals, variables, and expressions. Table 2.3 below describes one string function and six additional string methods where *str1* is the string "Python". Some further string methods will be presented in subsequent chapters.

TABLE 2.3 String operations (str1 = "Python").

Function or Method	Example	Value	Description
len	len(str1)	6	number of characters in the string
upper	str1.upper()	"PYTHON"	uppercases every alphabetical character
lower	str1.lower()	"python"	lowercases every alphabetical character
count	str1.count('th')	1	number of non-overlapping occurrences of the substring
capitalize	"coDE".capitalize()	"Code"	capitalizes the first letter of the string and lowercases the rest
title	"beN hur".title()	"Ben Hur"	capitalizes the first letter of each word in the string and lowercases the rest
rstrip	"ab ".rstrip()	"ab"	removes spaces from the right side of the string

Chained Methods

Consider the following two lines of code:

```
praise = "Good Doggie".upper()
numberOfGees = praise.count('G')
```

These two lines can be combined into the single line below that is said to **chain** the two methods.

```
numberOfGees = "Good Doggie".upper().count('G')
```

Chained methods are executed from left to right. Chaining often produces clearer code since it eliminates temporary variables, such as the variable *praise* above.

The *input* Function

The input function prompts the user to enter data. A typical input statement is

```
town = input("Enter the name of your city: ")
```

When Python reaches this statement, the string "Enter the name of your city: " is displayed and the program pauses. After the user types in the name of his or her city and presses the *Enter* (or *return*) key, the variable *town* is assigned the name of the city. (If the variable had not been created previously, it is created at this time.) The general form of an input statement is

```
variableName = input(prompt)
```

where *prompt* is a string that requests a response from the user.

Example 4 Parse a Name The following program requests a name from the user and then parses the name. When the program is run, the phrase "Enter a full name: " appears and execution of the program pauses. After the user types the words shown in black and presses the *Enter* (or *return*) key, the last two lines of output are displayed.

```
fullName = input("Enter a full name: ")
n = fullName.rfind(" ")
print("Last name:", fullName[n+1:])
print("First name(s):", fullName[:n])
```

[Run]

```
Enter a full name: Franklin Delano Roosevelt
Last name: Roosevelt
First name(s): Franklin Delano
```

The *int*, *float*, *eval*, and *str* Functions

If *str1* is a string containing a whole number, the **int** function will convert the string to an integer. If *str1* is a string containing any number, the **float** function will convert the string to a floating-point number. (The **float** function also converts an **integer** to a floating-point number.) If *str1* is a string consisting of a numeric expression, the **eval** function will evaluate the expression to an integer or floating-point number as appropriate.

Example 5 **Illustrate Functions** The following program illustrates the use of the **int**, **float**, and, **eval** functions.

```
print(int("23"))
print(float("23"))
print(eval("23"))
print(eval("23.5"))
x = 5
print(eval("23 + (2 * x)"))
```

[Run]

```
23
23.0
23
23.5
33
```

The **input** function always returns a string. However, a combination of an **input** function and an **int**, **float**, or **eval** function allows numbers to be input into a program. For instance, consider the following three statements:

```
age = int(input("Enter your age: "))
age = float(input("Enter your age: "))
age = eval(input("Enter your age: "))
```

Suppose the user responds with an integer, say 25. Then, after each of the statements above has been responded to, the statement `print(age)` would display 25, 25.0, and 25, respectively. However, if the user was a youngster, he or she might respond with the number 3.5. With the first input statement, a Traceback error message would result. After either the second or third input statement was executed, the **print** function would display 3.5. The **eval** function produced good results with either age.

The **int** and **float** functions execute faster than the **eval** function and are preferred by many Python programmers when they can be used safely. In this book we will use all three functions, but will favor the **eval** function.

The **int** and **float** functions also can be applied to appropriate numeric expressions. If *x* is an integer, the value of int(*x*) is *x*. If *x* is a floating-point number, the **int** function removes the decimal part of the number. The **float** function operates as expected. The **eval** function cannot be applied to numeric literals, variables, or expressions.

EXAMPLE	VALUE	EXAMPLE	VALUE
int(4.8)	4	float(4.67)	4.67
int(−4.8)	4	float(−4)	−4.0
int(4)	4	float(0)	0.0

The **str** function converts a number to its string representation. For instance, the value of `str(5.6)` is `"5.6"` and the value of `str(5.)` is `"5.0"`.

A string cannot be concatenated with a number. However, the invalid statement

```
strVar = numVar + '%'
```

can be replaced with the valid statement

```
strVar = str(numVar) + '%'
```

that concatenates two strings.

Internal Documentation

Program documentation is the inclusion of **comments** that specify the intent of the program, the purpose of the variables, and the tasks performed by individual portions of the program. To create a comment statement, begin a line with a number sign (#). Such a statement is completely ignored when the program is executed. Comments are sometimes called *remarks*. A line of code can be documented by adding a number sign, followed by the desired information, after the end of the line. Pressing Alt+3 and Alt+ 4 can be used in IDLE to comment and uncomment selected blocks of code.

Example 6 Parse a Name The following rewrite of Example 4 uses documentation. The first comment describes the entire program, the comment in the third line gives the meaning of a variable, and the final comment describes the purpose of the two lines that follow it.

```
## Break a name into two parts -- the last name and the first names.
fullName = input("Enter a full name: ")
n = fullName.rfind(" ")  # index of the space preceding the last name
# Display the desired information.
print("Last name:", fullName[n+1:])
print("First name(s):", fullName[:n])
```

Some of the benefits of documentation are as follows:

1. Other people can easily understand the program.
2. You can better understand the program when you read it later.
3. Long programs are easier to read because the purposes of individual pieces can be determined at a glance.

Good programming practice requires that programmers document their code while they are writing it. In fact, many software companies require a certain level of documentation before they release software and some judge a programmer's performance on how well their code is documented.

Line Continuation

A long statement can be split across two or more lines by ending each line (except the last) with a backslash character (\ ). For instance, the line

```
quotation = "Well written code is its own best documentation."
```

can be written as

```
quotation = "Well written code is its own " + \
            "best documentation."
```

Python has a feature that can be used to eliminate the need for line continuation with backslash characters. Any code enclosed in a pair of parentheses can span multiple lines. Since any expression can be enclosed in parentheses, this feature can nearly always be used. For instance, the statement above can be written as

```
quotation = ("Well written code is its own " +
             "best documentation.")
```

This method of line continuation has become the preferred style for most Python programmers and will be used whenever possible in this textbook.

Indexing and Slicing Out of Bounds

Python does not allow out of bounds indexing for individual characters of strings, but does allow out of bounds indices for slices. For instance, if

```
str1 = "Python"
```

then `print(str1[7])` and `print(str1[-7])` trigger the Traceback error message *IndexError*.

If the left index in a slice is too far negative, the slice will start at the beginning of the string, and if the right index is too large, the slice will go to the end of the string. For instance,

`str1[-10:10]` is `"Python"`
`str1[-10:3]` is `"Pyt"`
`str1[2:10]` is `"thon"`

Comments

1. In this textbook, we usually surround one-character strings with single quotation marks and all other strings with double quotation marks.
2. Since a string expression is any combination of literals, variables, functions, methods, and operators that can be evaluated to produce a string, a single string or variable is a special case of an expression.
3. Every character in a string has two indices—one positive and one negative. Therefore, the numbers *m* and *n* in an expression of the form strValue[*m*:*n*] can have opposite signs. If the character having index *m* is to the left of the character having index *n*, then the slice will consist of the substring beginning with the character having index *m* and

ending with the character to the left of the character having index n. For instance, the value of "Python"[−4:5] is "tho". Of course, if the character having index m is not to the left of the character having index n, then the slice will be the empty string.

4. Individual characters within a string cannot be changed directly. For instance, the code below, which intends to change the word *resort* to the word *report*, produces a Traceback error message.

```
word = "resort"
word[2] = 'p'
```

5. The operator += performs an augmented concatenation assignment for strings.
6. IDLE displays strings in the color green and comments in the color red.
7. Method names, like names of variables and functions, are case-sensitive.
8. For readability purposes, you should not chain more than three methods together.
9. Strings are said to have type **str**. The statement `print(dir(str))` displays all the string methods. (Ignore the items that begin and end with double underscore characters.)

Practice Problems 2.2

1. Assuming that $0 \leq m \leq n \leq \text{len}(str1)$, how many characters are in `str1[m:n]`?
2. What is displayed by the statement `print("Computer".find('E'))`?

EXERCISES 2.2

In Exercises 1 through 4, determine the output displayed by the lines of code.

1. `print("Python")`
2. `print("Hello")`
3.
```
var = "Ernie"
print(var)
```
4.
```
var = "Bert"
print(var)
```

In Exercises 5 through 46, determine the value of the expression.

5. `"Python"[4]`
6. `"Python"[-2]`
7. `"Hello Python!"[-9]`
8. `"Python"[5]`
9. `"Python"[0:3]`
10. `"Python"[2:2]`
11. `"Python"[:2]`
12. `"Python"[2:]`
13. `"Python"[-3:-2]`
14. `"Python"[-5:-1]`
15. `"Python"[2:-2]`
16. `"Python"[-4:4]`
17. `"Python"[:]`
18. `"Python"[-10:10]`
19. `"Python".find("tho")`
20. `"Python".find("ty")`
21. `"Python".find("oh")`
22. `"Python".find("Pyt")`
23. `"whizzbuzz".rfind("zz")`
24. `"whizzbuzz".find("zz")`
25. `" Python".lstrip()`
26. `"hello_world".startswith("hell")`
27. `"smallElements".capitalize()`
28. `"hello_python".rpartition('_')`
29. `"PyThOn'.swapcase()`
30. `"python/java/c++".split('/')`

31. `"8 Ball".title()`
32. `len("brrr")`
33. `"8 Ball".upper()`
34. `"whippersnapper".count("pp")`
35. `"Python"[-1*len("Python")-1:3]`
36. `"Python".lower()`
37. `"the artist".title()`
38. `len("Gravity   ".rstrip())`
39. `len("Grand Hotel"[:6].rstrip())`
40. `"king lear".title()`
41. `"let it go".title().find('G')`
42. `"Hello World!".lower().find('wo')`
43. `"Amazon".lower().count('a')`
44. `"Python".upper().find("tho")`
45. `"john's school".capitalize()`
46. `"all clear".title().count('a')`

In Exercises 47 through 70, determine the output displayed by the lines of code.

47.
```
a = 4
b = 6
c = "Municipality"
d = "pal"
print(len(c))
print(c.upper())
print(c[a:b] + c[b + 4:])
print(c.find(d))
```

48.
```
m = 4
n = 3
s = "Microsoft"
t = "soft"
print(len(s))
print(s.lower())
print(s[m:m + 2])
print(s.find(t))
```

49. `print("f" + "lute")`
50. `print("a" + "cute")`
51. `print("Your age is " + str(21) + ".")`
52. `print("Fred has " + str(2) + " children.")`
53.
```
r = "A ROSE"
b = " IS "
print(r + b + r + b + r)
```
54.
```
sentence = "ALPHONSE TIPPYTOED AWAY."
print(sentence[12:15] + sentence[3:6])
```
55.
```
var = "WALLA"
var += 2 * var
print(var)
```
56.
```
str1 = "mur"
str1 += str1
print(str1)
```
57.
```
str1 = "good"
str1 += "bye"
print(str1)
```
58.
```
var = "eight"
var += "h"
print(var)
```
59. `print('M' + ('m' * 3) * 2 + '.')`
60. `print(('*' * 3) + "YES" + ('*' * 3))`
61. `print('a' + (" " * 5) + 'b')`
62. `print("spam" * 4)`
63.
```
s = "trom"
n = 76
print(n, s + "bones")
```
64.
```
str1 = "5"
num = 0.5 + int(str1)
print(num)
```
65.
```
num = input("Enter an integer: ")
print('1' + str(num))
```
(Assume the response is 7.)

66.
```
num = int(input("Enter an integer: "))
print(1 + num)
```
(Assume the response is *7*.)

67.
```
num = float(input("Enter a number: "))
print(1 + num)
```
(Assume the response is *7*.)

68.
```
num = eval(input("Enter a number: "))
print(1 + num)
```
(Assume the response is *7*.)

69.
```
film = "the great gatsby".title()[:10].rstrip()
print(film, len(film))
```

70.
```
batmanAndRobin = "THE DYNAMIC DUO".lower().title()
print(batmanAndRobin)
```

71. Give a simple expression that lops off the last character of a string.

72. Give a simple expression that lops off the first character of a string.

73. What is the negative index of the first character in a string of eight characters?

74. What is the positive index of the last character in a string of eight characters?

75. (True or false) If *n* is the length of *str1*, then `str1[n - 1:]` is the string consisting of the last character of *str1*.

76. (True or false) If *n* is the length of *str1*, then `str1[n - 2:]` is the string consisting of the last two characters of *str1*.

77. (True or false) `str1[:n]` consists of the first *n* characters of *str1*.

78. (True or false) `str1[-n:]` consists of the last *n* characters of *str1*.

In Exercises 79 through 92, identify all errors.

79.
```
phoneNumber = 234-5678
print("My phone number is " + phoneNumber)
```

80.
```
quote = I came to Casablanca for the waters.
print(quote + ": " + "Bogart")
```

81.
```
for = "happily ever after."
print("They lived " + for)
```

82.
```
age = input("Enter your age: ")
print("Next year you will be " + (age + 1))
```

83.
```
print('Say it ain't so.')
```

84.
```
print("George "Babe" Ruth")
```

85.
```
print("Python".UPPER())
```

86.
```
print("Python".lower)
```

87.
```
age = 19
print("Age: " + age)
```

88.
```
num = 1234
print(num[3])
```

89.
```
num = '1234'
print(num.find(2))
```

90.
```
num = 45
print(len(num))
```

91.
```
language = "Python"
language[4] = 'r'
```

92.
```
show = "Spamalot"
print(show[9])
```

In Exercises 93 through 96, write a program having one line for each step. Lines that display data should use the given variable names.

93. Inventor The following steps give the name and birth year of a famous inventor.

(a) Create the variable *firstName* and assign it the value "Thomas".
(b) Create the variable *middleName* and assign it the value "Alva".
(c) Create the variable *lastName* and assign it the value "Edison".
(d) Create the variable *yearOfBirth* and assign it the value 1847.
(e) Display the phrase "The year of birth of" followed by the inventor's full name, followed by "is", and the inventor's year of birth.

94. Price of Ketchup The following steps compute the price of ketchup.

(a) Create the variable *item* and assign it the value "ketchup".
(b) Create the variable *regularPrice* and assign it the value 1.80.
(c) Create the variable *discount* and assign it the value .27.
(d) Display the phrase "1.53 is the sale price of ketchup."

95. Copyright Statement The following steps display a copyright statement.

(a) Create the variable *publisher* and assign it the value "Pearson".
(b) Display the phrase "(c) Pearson".

96. Advice The following steps give advice.

(a) Create the variable *prefix* and assign it the value "Fore".
(b) Display the phrase "Forewarned is Forearmed."

***Note:* For each of the following exercises, a possible output is shown in a shaded box. Responses to input statements appear underlined.**

97. Distance from a Storm If n is the number of seconds between lightning and thunder, the storm is $n/5$ miles away. Write a program that requests the number of seconds between lightning and thunder and reports the distance from the storm rounded to two decimal places. See Fig. 2.7.

```
Enter number of seconds between
lightning and thunder: 1.25
Distance from storm: 0.25 miles.
```

FIGURE 2.7 Possible outcome of Exercise 97.

```
Enter your age: 20
Enter your resting heart rate: 70
Training heart rate: 161 beats/min.
```

FIGURE 2.8 Possible outcome of Exercise 98.

98. Training Heart Rate The American College of Sports Medicine recommends that you maintain your *training heart rate* during an aerobic workout. Your training heart rate is computed as $.7 * (220 - a) + .3 * r$, where a is your age and r is your resting heart rate (your pulse when you first awaken). Write a program to request a person's age and resting heart rate and display their training heart rate. See Fig. 2.8.

99. Triathlon The number of calories burned per hour by cycling, running, and swimming are 200, 475, and 275, respectively. A person loses 1 pound of weight for each 3,500 calories burned. Write a program to request the number of hours spent at each activity and then display the number of pounds worked off. See Fig. 2.9.

```
Enter number of hours cycling: 2
Enter number of hours running: 3
Enter number of hours swimming: 1
Weight loss: 0.6 pounds
```

FIGURE 2.9 Possible outcome of Exercise 99.

```
Enter wattage: 100
Enter number of hours used: 720
Enter price per kWh in cents: 11.76
Cost of electricity: $6.12
```

FIGURE 2.10 Possible outcome of Exercise 100.

100. Cost of Electricity The cost of the electricity used by a device is given by the formula

$$\text{cost of electricity (in dollars)} = \frac{\text{wattage of device} \cdot \text{hours used}}{1{,}000 \cdot \text{cost per kWh (in cents)}}$$

where kWh is an abbreviation for "kilowatt hour." The cost per kWh of electricity varies with locality. Suppose the current average cost of electricity for a residential customer in the United States is 11.76¢ per kWh. Write a program that allows the user to calculate the cost of operating an electrical device. Figure 2.10 calculates the cost of keeping a light bulb turned on for an entire month.

101. Baseball Write a program to request the name of a baseball team, the number of games won, and the number of games lost as input, and then display the name of the team and the percentage of games won. See Fig. 2.11.

```
Enter name of team: Yankees
Enter number of games won: 84
Enter number of games lost: 78
Yankees won 51.9% of their games.
```

FIGURE 2.11 Possible outcome of Exercise 101.

```
Enter earnings per share: 5.25
Enter price per share: 68.25
Price-to-Earnings ratio: 13.0
```

FIGURE 2.12 Possible outcome of Exercise 102.

102. Price-to-Earnings Ratio Write a program that requests a company's earnings-per-share for the year and the price of one share of stock as input, and then displays the company's price-to-earnings ratio (that is, price ÷ earnings). See Fig. 2.12.

103. Car Speed The formula $s = \sqrt{24d}$ gives an estimate of the speed in miles per hour of a car that skidded d feet on dry concrete when the brakes were applied. Write a program that requests the distance skidded and then displays the estimated speed of the car. See Fig. 2.13. ***Note:*** $\sqrt{x} = x^{.5}$.

```
Enter distance skidded: 54
Estimated speed: 36.0 miles per hour
```

FIGURE 2.13 Possible outcome of Exercise 103.

```
Enter percentage: 125%
Equivalent decimal: 1.25
```

FIGURE 2.14 Possible outcome of Exercise 104.

104. Percentages Write a program that converts a percentage to a decimal. See Fig. 2.14.

105. Convert Speeds On May 6, 1954, British runner Sir Roger Bannister became the first person to run the mile in less than 4 minutes. His average speed was 24.20 kilometers per hour. Write a program that requests a speed in kilometers per hour as

input and then displays the speed in miles per hour. See Fig. 2.15. ***Note:*** One kilometer is .6214 of a mile.

```
Enter speed in KPH: 24.20
Speed in MPH: 15.04
```

FIGURE 2.15 Possible outcome of Exercise 105.

```
Enter amount of bill: 21.50
Enter percentage tip: 18
Tip: $3.87
```

FIGURE 2.16 Possible outcome of Exercise 106.

106. Server's Tip Write a program that calculates the amount of a server's tip, given the amount of the bill and the percentage tip as input. See Fig. 2.16.

107. Equivalent Interest Rates Interest earned on municipal bonds from an investor's home state is not taxed, whereas interest earned on CDs *is* taxed. Therefore, in order for a CD to earn as much as a municipal bond, the CD must pay a higher interest rate. How much higher the interest rate must be depends on the investor's tax bracket. Write a program that requests a tax bracket and a municipal bond interest rate as input, and then displays the CD interest rate having the same yield. See Fig. 2.17. ***Note:*** If the tax bracket is expressed as a decimal, then

$$\text{CD interest rate} = \frac{\text{municipal bond interest rate}}{(1 - \text{tax bracket})}.$$

```
Enter tax bracket (as decimal): .37
Enter municipal bond interest rate (as %): 3.26
Equivalent CD interest rate: 5.175%
```

FIGURE 2.17 Possible outcome of Exercise 107.

108. Marketing Terms The *markup* of an item is the difference between its *selling price* and its *purchase price*. Two other marketing terms are

$$\text{percentage markup} = \frac{\text{markup}}{\text{purchase price}} \quad \textit{and} \quad \text{profit margin} = \frac{\text{markup}}{\text{selling price}}$$

where the quotients are expressed as percentages. Write a program that computes the markup, percentage markup, and profit margin of an item. See Fig. 2.18. Notice that when the purchase price is tripled, the percentage markup is 200%.

```
Enter purchase price: 215
Enter selling price: 645
Markup: $430.0
Percentage markup: 200.0%
Profit margin: 66.67%
```

FIGURE 2.18 Possible outcome of Exercise 108.

```
Enter number: 123.45678
3 digits to left of decimal point
5 digits to right of decimal point
```

FIGURE 2.19 Possible outcome of Exercise 109.

109. Analyze a Number Write a program that requests a positive number containing a decimal point as input and then displays the number of digits to the left of the decimal point and the number of digits to the right of the decimal point. See Fig. 2.19.

110. Word Replacement Write a program that requests a sentence, a word in the sentence, and another word and then displays the sentence with the first word replaced by the second. See Fig. 2.20.

```
Enter a sentence: What you don't know won't hurt you.
Enter word to replace: know
Enter replacement word: owe
What you don't owe won't hurt you.
```

FIGURE 2.20 Possible outcome of Exercise 110.

111. Convert Months Write a program that asks the user to enter a whole number of months as input and then converts that amount of time to years and months. See Fig. 2.21. The program should use both integer division and the modulus operator.

```
Enter number of months: 234
234 months is 19 years and 6 months.
```

FIGURE 2.21 Possible outcome of Exercise 111.

```
Enter number of inches: 185
185 inches is 15 feet and 5 inches.
```

FIGURE 2.22 Possible outcome of Exercise 112.

112. Convert Lengths Write a program that asks the user to enter a whole number of inches and then converts that length to feet and inches. See Fig. 2.22. The program should use both integer division and the modulus operator.

Solutions to Practice Problems 2.2

1. $n - m$. When $m = 0$ the number of characters in str1[0:n] is n. Increasing the number 0 to m, decreases the number of characters by m.
2. −1. There is no uppercase letter *E* in the string "Computer". The find method distinguishes between uppercase and lowercase letters.

2.3 Output

Enhanced output can be produced by the print function with two optional arguments and the use of the format method.

Optional *print* Argument *sep*

A statement of the form

```
print(value0, value1, . . ., valueN)
```

where the values are strings or numbers, displays the values one after another with successive values separated by a space. We say that the print function uses the string consisting of one space character as a **separator**. We can optionally change the separator to any string we like with the **sep argument**. If *sepString* is a string, then a statement of the form

```
print(value0, value1, . . ., valueN, sep=sepString)
```

displays the values with successive values separated by *sepString*. Some examples are as follows:

STATEMENT	OUTCOME
`print("Hello", "World!", sep="**")`	`Hello**World!`
`print("Hello", "World!", sep="")`	`HelloWorld!`
`print("1", "two", 3, sep="   ")`	`1   two   3`

Optional *print* Argument *end*

After any of the statements above are executed, the display of output on the current line comes to an end, and the next **print** statement will display its output on the next line. We say that the **print** statement ends by executing a **newline operation**. (We also say that the **print** statement moved the cursor to the beginning of the next line or that the **print** statement performed a "carriage return and line feed.") We can optionally change the ending operation with the **end argument**. If *endString* is a string, then a statement of the form

```
print(value0, value1, . . ., valueN, end=endString)
```

displays *value0* through *valueN* and then displays *endString* on the same line, without performing a newline operation. Here are some lines of code that use the **end** argument.

```
print("Hello", end=" ")
print("World!")
```

[Run]

```
Hello World!
```

```
print("Hello", end="")
print("World!")
```

[Run]

```
HelloWorld!
```

Escape Sequences

Escape sequences are short sequences that are placed in strings to instruct the cursor or to permit some special characters to be printed. The first character is always a backslash (\). The two most common cursor-instructing escape sequences are **\t** (induces a horizontal tab) and **\n** (induces a newline operation). By default, the tab size is eight spaces, but can be increased or decreased with the **expandtabs method**.

Example 1 Escape Sequences The following program demonstrates the use of the escape sequences \t and \n.

```
## Demonstrate use of escape sequences.
print("01234567890123456")
print("a\tb\tc")
print("a\tb\tc".expandtabs(5))
print("Nudge, \tnudge, \nwink, \twink.".expandtabs(11))
```

[Run]

```
01234567890123456
a       b       c
a    b    c
Nudge,     nudge,
wink,      wink.
```

Each escape sequence is treated as a single character when determining the length of a string. For instance, `len("a\tb\tc")` has value 5. The backslash is not considered to be a character, but rather an indicator telling Python to treat the character following it in a special way. The escape sequence \n is often referred to as the **newline character**.

The backslash also can be used to treat quotation marks as ordinary characters. For instance, the statement `print('Say it ain\'t so.')` displays the third word as *ain't*. The backslash character tells Python to treat the quotation mark as an ordinary single quotation mark and not as a surrounding quotation mark. Two other useful escape sequences are \" and \\ which cause the print function to display a double quotation mark and a backslash character, respectively.

In future chapters we frequently encounter strings that end with a newline character. For instance, each line of a text file is a string ending with a newline character. The string method rstrip can be used to remove newline characters from the ends of strings. For instance, if *str1* has the value "xyz\n", then `str1.rstrip()` will have the value "xyz". Also, when the int, float, and eval functions are evaluated at a string ending with a newline character, they ignore the newline character. For instance, `int('7\n')` has the same value as `int('7')`.

Justifying Output in a Field

VideoNote
Print Formatting

Programs often display output in columns of a fixed width. The methods **ljust(n)**, **rjust(n)**, and **center(n)** can be used to left-justify, right-justify, and center string output in a field of width *n*. If the string does not use the entire width of the field, the string is padded on the right, left, or both sides with spaces. If the string is longer than the allocated width, the justification method is ignored.

Example 2 Justifying Output The following program uses the three justification methods to create a table of the top three home run hitters in professional baseball. The first line was added to identify the columns of the table. The first five columns (columns 0 through 4) list the ranks of the top three hitters. The numbers 1, 2, and 3 are each centered in a field of width 5. The next 20 columns (columns 5 through 24) hold the names of the top three hitters, with each name left justified in a field of width 20. Each name is padded on the right with space characters. The last three columns (columns 25 through 27) hold the number of home runs hit by the players. Since each of the numbers is three digits long, they exactly fill the field of width 3 set aside for them. The output for this column would be the same even if the rjust method was not used.

```
## Demonstrate justification of output.
print("01234567890123456789012345678901234567")
print("Rank".ljust(5), "Player".ljust(20), "HR".rjust(3), sep="")
print('1'.center(5), "Barry Bonds".ljust(20), "762".rjust(3), sep="")
print('2'.center(5), "Hank Aaron".ljust(20), "755".rjust(3), sep="")
print('3'.center(5), "Babe Ruth".ljust(20), "714".rjust(3), sep="")
```

[Run]

```
01234567890123456789012345678901234567
Rank Player               HR
  1  Barry Bonds          762
  2  Hank Aaron           755
  3  Babe Ruth            714
```

Justifying Output with the *format* Method

The **format method** is a fairly recent addition to Python that can perform the same tasks as the justification methods and much more. For instance, it can place thousands separators in numbers, round numbers, and convert numbers to percentages. We will begin by demonstrating the method's justification capabilities and then present some of its other features.

If *str1* is a string and *w* is a field width, then statements of the forms

```
print("{0:<ws}".format(str1))
print("{0:^ws}".format(str1))
print("{0:>ws}".format(str1))
```

produce the same output as the statements

```
print(str1.ljust(w))
print(str1.center(w))
print(str1.rjust(w))
```

If *num* is a number and *w* is a field width, then statements of the forms

```
print("{0:<wn}".format(num))
print("{0:^wn}".format(num))
print("{0:>wn}".format(num))
```

produce the same output as the statements

```
print(str(num).ljust(w))
print(str(num).center(w))
print(str(num).rjust(w))
```

Notice that the format method accepts numbers directly; they do not have to be converted to strings. The symbols <, ^, and > that precede the width of each field instruct the print function to left-justify, center, and right-justify, respectively.

In each of the statements above containing the format method, there is a single argument (*num*) in the format method. Often there are several arguments, referred to by positions counting from zero. The 0 before the colon in the curly braces refers to the fact that *num* is in the 0^{th} position. When there are several arguments, there are several pairs of curly braces, with each pair of curly braces associated with an argument. The numbers preceding the colons inside each pair of curly braces give the position of the argument it formats.

Example 3 **Justifying Output** The following program produces the same output as Example 2, but using the format method. Consider the fourth line. The formatting braces {0:^5n}, {1:<20s}, and {2:>3n} determine the formatting of the number 1, the string "Barry Bonds", and the number 762, respectively.

```
## Demonstrate justification of output.
print("01234567890123456789012345678")
print("{0:^5s}{1:<20s}{2:>3s}".format("Rank", "Player", "HR"))
print("{0:^5n}{1:<20s}{2:>3n}".format(1, "Barry Bonds", 762))
print("{0:^5n}{1:<20s}{2:>3n}".format(2, "Hank Aaron", 755))
print("{0:^5n}{1:<20s}{2:>3n}".format(3, "Babe Ruth", 714))
```

When numbers are being formatted, rather than using the letter *n* inside the curly braces, which corresponds to any type of number, we use the letter *d* for integers, the letter *f*

for floating-point numbers, and the symbol % for numbers to be displayed as percentages. When *f* and % are used, they should be preceded by a period and a whole number. The whole number determines the number of decimal places to be displayed. In each of the three cases, we also can specify if we want thousands separators by inserting a comma after the field-width number.

When the format method is used to format a number, *right-justify* is the default justification. Therefore, when none of the symbols <, ^, or > are present, the number will be displayed right-justified in its field. Table 2.4 shows some statements and the outcomes they produce.

TABLE 2.4 Demonstrate number formatting.

Statement	Outcome	Comment
`print("{0:10d}".format(12345678))`	`12345678`	number is an integer
`print("{0:10,d}".format(12345678))`	`12,345,678`	thousands separators added
`print("{0:10.2f}".format(1234.5678))`	`1234.57`	rounded
`print("{0:10,.2f}".format(1234.5678))`	`1,234.57`	rounded and separators added
`print("{0:10,.3f}".format(1234.5678))`	`1,234.568`	rounded and separators added
`print("{0:10.2%}".format(12.345678))`	`1234.57%`	changed to % and rounded
`print("{0:10,.3%}".format(12.34569))`	`1,234.568%`	%, rounded, separators

The field-width number following the colon can be omitted. If so, the number is displayed (without any alignment) as determined by the other specifiers following the colon.

So far, the string preceding "`.format`" has consisted of one or more pairs of curly braces. However, the string can be any string containing curly braces. In that case, the curly braces are placeholders telling Python where to insert the arguments from the format method.

Example 4 State Data The following program demonstrates the use of placing curly braces inside a string.

```
## Demonstrate use of the format method.
print("The area of {0:s} is {1:,d} square miles.".format("Texas", 268820))
str1 = "The population of {0:s} is {1:.2%} of the U.S. population."
print(str1.format("Texas", 26448000 / 309000000))
```

[Run]

```
The area of Texas is 268,820 square miles.
The population of Texas is 8.56% of the U.S. population.
```

Comments

1. When the right side of the colon in a pair of curly braces is just the letter *s*, the colon and the letter *s* can be omitted. For instance, {0:s} can abbreviated to {0}. A placeholder such as {0} applies not only to strings, but also to numbers and expressions.
2. When the format method is used to format a string, left-justify is the default justification. Therefore, when a <, ^, or > symbol is not present, the string will be displayed left-justified in its field.
3. The rstrip method not only removes newline characters from the end of a string, but removes all ending spaces and escape sequences. When the int, float, or eval function is applied to a string, it ignores all spaces and escape sequences at the end of the string.

4. A common error is to write an escape sequence with a forward-slash (/) instead of the backslash (\), the proper character.

Practice Problems 2.3

Determine the output displayed by the lines of code.

1.
```
print("{0:s} and {1:s}".format("spam", "eggs"))
```
2.
```
str1 = "Ask not what {0:s} {1:s} you, ask what you {1:s} {0:s}."
print(str1.format("your country", "can do for"))
```
3. Rewrite the following statement without using escape sequences.
```
print("He said \"How ya doin?\" to me.")
```

EXERCISES 2.3

In Exercises 1 through 50, determine the output displayed by the lines of code.

1.
```
print("merry", " christmas", '!', sep="")
```
2.
```
print("Price: ", '$', 23.45, sep="")
```
3.
```
print("Portion: ", 90, '%', sep="")
```
4.
```
print("Py", "th", "on", sep="")
```
5.
```
print(1, 2, 3, sep=" x ")
```
6.
```
print("tic", "tac", "toe", sep='-')
```
7.
```
print("father", "in", "law", sep='-')
```
8.
```
print("one", " two", " three", sep=',')
```
9.
```
print("What is your name",
end = '?\n'))
print("John")
```
10.
```
print("spam", end=" and ")
print("eggs")
```
11.
```
print("Py", end="")
print("thon")
```
12.
```
print("on","site",sep='-',end="")
print("repair")
```
13.
```
print("Hello\n")
print("World!")
```
14.
```
print("Hello\n" , end = ',')
print("World!")
```
15.
```
print("One\t\tTwo\n--Three-Four")
```
16.
```
print("1\t2\t3")
print("\tDetroit\tLions")
print("Indiana\t\tColts")
```
17.
```
print("NUMBER\tSQUARE\tCUBE")
print(str(2) + "\t" + str(2 ** 2) + "\t" + str(2 ** 3))
print(str(3) + "\t" + str(3 ** 2) + "\t" + str(3 ** 3))
```
18.
```
print("COUNTRY\t", "LAND AREA")
print("India\t", 2.5, "million sq km")
print("China\t", 9.6, "million sq km")
```
19.
```
print("Hello\tWorld!")
print("Hello\t\tWorld!".expandtabs(2))
```
20.
```
print("STATE\tCAPITAL".expandtabs(15))
print("North Dakota\tBismarck".expandtabs(15))
print("South Dakota\tPierre".expandtabs(15))
```

21.
```
print("012345.67890")
print("A ".rjust(5), " B ".center(5),"  C".ljust(5), sep="|")
```

22.
```
print("0123456789012345")
print("one".center(7), "two".ljust(4), "three".rjust(6), sep="")
```

23.
```
print("0123456789012345")
print("{0:^7s}{1:5s}{2:>7s}".format("one", "two", "three"))
```

24.
```
print("01234567890")
print("{0:>5s}{1:^5s}{2:5s}".format("A", "B", "C"))
```

25.
```
print("0123456789")
print("{0:10.1%}".format(.123))
print("{0:^10.1%}".format(1.23))
print("{0:<10,.1%}".format(12.3))
```

26.
```
print("0123456789")
print("{0:10,d}".format(1234))
print("{0:^10,d}".format(1234))
print("{0:<10,d}".format(1234))
```

27.
```
print("${0:,.1f}".format(1234.567))
```

28.
```
print("{0:,.0f}".format(1234.567))
```

29.
```
print("{0:,.1f}".format(1.234))
```

30.
```
print("${0:,.2f}".format(1234))
```

31.
```
print("{0:10s}{1:^16s} {2:s}".format("Team", "Fifa points",
                                     "% fans of World Pop."))
print("{0:10s}{1:^16,d}{2:10.2%}".format("Germany", 1725,.3412))
print("{0:10s}{1:^16,d}{2:10.2%}".format("Argentina", 1538,.25851))
print("{0:10s}{1:^16,d}{2:10.2%}".format("Columbia", 1450,.25523))
```

32.
```
print("{0:14s}{1:s}".format("Major", "Percent of Students"))
print("{0:14s}{1:10.1%}".format("Biology", .062))
print("{0:14s}{1:10.1%}".format("Psychology", .054))
print("{0:14s}{1:10.1%}".format("Nursing", .047))
```

33.
```
print("When nothing goes {0:s} go {1:s}.".format("right", "left"))
```

34.
```
print("Plan {0:s}, code {1:s}.".format("first", "later"))
```

35.
```
print("{0:s} are the {1:s} of {0:s}r own destiny".format("you","creator"))
```

36.
```
print("And now for {0:s} completely {1:s}.".format("something",
"different"))
```

37.
```
x=3
y = 4
print("The matrix of {0:d} and {1:d} has {2:d} elements.".format(x, y, x * y))
```

38.
```
str1 = "{0:s} has {1:.1f} billion users in the world."
print(str1.format("Facebook", 1.3))
```

39.
```
x = 2    # square root of 2 is 1.414213562 to 9 decimal places
print("The square root of {0:n} is about {1:.4f}.".format(x, x ** .5))
```

40.
```
pi = 3.14159265898 # to 11 decimal places
print("Pi is approximately {0:.3f}.".format(pi))
```

41.
```
str1 = "In a randomly selected group of {0:d} people, the " + \
       "probability\nis {1:.2f} that 2 people have the same birthday."
print(str1.format(23, .507397))
```

42.
```
# Population Survey of Canada in 2014
areaOfCanada = 9984670
popOfCanada = 35344962 #35344962/9984670 is 3.539922902 to 9 decimal places
str1 = "The population of Canada is ${0:.3f} per km square."
print(str1.format(costOfCanada / areaOfCanada))
```

43.
```
str1 = "You miss {0:.0%} of the shots you never take. - Wayne Gretsky"
print(str1.format(1))
```

44.
```
str1 = "{0:.0%} of the members of the U.S. Senate are from {1:s}."
print(str1.format(12 / 100, "New England"))
```

45.
```
# 43/193 is .2227979275 to 10 decimal places
print("{0:.2%} of the UN nations are in {1:s}.".format(43/193, "Europe"))
```

46.
```
# 9984670/3794000 is 2.631700053 to 9 decimal places
str1 = "The area of {0:s} is {1:.1%} of the area of the U.S."
print(str1.format("Canada", 9984670 / 3794000))
```

47.
```
print("{0:s}{1:s}{0:s}".format("abra", "cad"))
```

48.
```
print("When you have {0:s} to {1:s}, {1:s} {0:s}.".format("nothing", "say"))
```

49.
```
str1 = "Be {0:s} whenever {1:s}. It is always {1:s}. - Dalai Lama"
print(str1.format("kind", "possible"))
```

50.
```
str1 = "If {0:s} dream it, {0:s} do it. - Walt Disney"
print(str1.format("you can"))
```

51. Do `print("Hello")` and `print("Hello", end="\n")` produce the same output?

52. Do `print("Hello\tWorld!")` and `print("Hello\tWorld!".expandtabs(8))` produce the same output?

In Exercises 53 through 58, write a program to carry out the stated task.[2]

53. **Server's Tip** Calculate the amount of a server's tip, given the amount of the bill and the percentage tip as input. See Fig. 2.23.

```
Enter amount of bill: 21.50
Enter percentage tip: 18
Tip: $3.87
```

FIGURE 2.23 Possible outcome of Exercise 53.

```
Enter revenue: 550000
Enter expenses: 410000
Net income: $140,000.00
```

FIGURE 2.24 Possible outcome of Exercise 54.

54. **Income** Request a company's annual revenue and expenses as input, and display the company's net income (revenue minus expenses). See Fig. 2.24.

55. **Change in Salary** A common misconception is that if you receive a 10% pay raise and later a 10% pay cut, your salary will be unchanged. Request a salary as input and then display the salary after receiving a 10% pay raise followed by a 10% pay cut. The program also should display the percentage change in salary. See Fig. 2.25.

[2]For each of the following exercises, a possible output is shown in a shaded box. Responses to input statements appear underlined.

```
Enter beginning salary: 35000
New salary: $34,650.00
Change: -1.00%
```

FIGURE 2.25 Possible outcome of Exercise 55.

```
Enter beginning salary: 35000
New salary: $40,516.88
Change: 15.76%
```

FIGURE 2.26 Possible outcome of Exercise 56.

56. Change in Salary A common misconception is that if you receive three successive 5% pay raises, then your original salary will have increased by 15%. Request a salary as input and then display the salary after receiving three successive 5% pay raises. The program also should display the percentage change in salary. See Fig. 2.26.

57. Future Value If P dollars (called the *principal*) is invested at r% interest compounded annually, then the future value of the investment after n years is given by the formula

$$\text{future value} = P\left(1 + \frac{r}{100}\right)^n.$$

Calculate the future value of an investment after the user enters the principal, interest rate, and number of years. Figure 2.27 shows that $1,000 invested at 5% interest will grow to $1,157.63 in 3 years.

```
Enter principal: 1000
Enter interest rate (as %): 5
Enter number of years: 3
Future value: $1,157.63
```

FIGURE 2.27 Possible outcome of Exercise 57.

```
Enter future value: 10000
Enter interest rate (as %): 4
Enter number of years: 6
Present value: $7,903.15
```

FIGURE 2.28 Possible outcome of Exercise 58.

58. Present Value The present value of f dollars at interest rate r% compounded annually for n years is the amount of money that must be invested now in order to grow to f dollars (called the *future value*) in n years where the interest rate is r% per year. The formula for present value is

$$\text{present value} = \frac{f}{\left(1 + \frac{r}{100}\right)^n}.$$

Calculate the present value of an investment after the user enters the future value, interest rate, and number of years. Figure 2.28 shows that at 4% interest per year, $7,903.15 must be invested now in order to have $10,000 after 6 years.

Solutions to Practice Problems 2.3

1. **`spam and eggs`**. The **`s`** specifier in the curly braces is the default specifier. Therefore, the `print` statement could have been written

   ```
   print("{0} and {1}".format("spam", "eggs"))
   ```

   We will use the **`s`** specifier in our programs since it improves readability. It reminds the programmer that a string is required as the argument in the set of arguments.

2. **`Ask not what your country can do for you, ask what you can do for your country.`**

The strings requested by the first two sets of curly braces are obvious. The third set of curly braces begins with `1` and therefore is requesting the argument in position 1, namely `"can do for"`. Similarly, the fourth set of curly braces is requesting the argument in position 0. The ability to use arguments more than once is a nice feature of the format method.

3. `print('He said "How ya doin?" to me.')`

2.4 Lists, Tuples, and Files—An Introduction

The Python documentation and this textbook use the term **object** to refer to any instance of a data type. Python's core objects are numbers, strings, lists, tuples, files, sets, and dictionaries. We have already discussed numbers and strings. In this section we discuss lists, tuples, and files. Sets and dictionaries are discussed in Chapter 5.


VideoNote
The *list* Object

The *list* Object

A **list** is an ordered sequence of Python objects. The objects can be of any type and do not have to all be the same type.

A list is constructed by writing its items enclosed in square brackets, with the items separated by commas. Some examples of lists are

```
["Seahawks", 2014, "CenturyLink Field"]
[5, 10, 4, 5]
["spam", "ni"]
```

Lists are usually assigned to a name. For instance, we might write

```
team = ["Seahawks", 2014, "CenturyLink Field"]
nums = [5, 10, 4, 5]
words = ["spam", "ni"]
```

TABLE 2.5 List operations. (The lists *team*, *nums*, and *words* are given above.)

Function or Method	Example	Value	Description
len	len(words)	2	number of items in list
max	max(nums)	10	greatest (items must have same type)
min	min(nums)	4	least (items must have same type)
sum	sum(nums)	24	total (items must be numbers)
count	nums.count(5)	2	number of occurrences of an object
index	nums.index(4)	2	index of first occurrence of an object
reverse	words.reverse()	["ni", "spam"]	reverses the order of the items
clear	team.clear()	[]	[] is the empty list
append	nums.append(7)	[5, 10, 4, 5, 7]	inserts object at end of list
extend	nums.extend([1, 2])	[5, 10, 4, 5, 1, 2]	inserts new list's items at end of list
del	del team[−1]	["Seahawks", 2014]	removes item with stated index
remove	nums.remove(5)	[10, 4, 5]	removes first occurrence of an object
insert	words.insert(1, "wink")	["spam", "wink", "ni"]	insert new item before item of given index
+	['a', 1] + [2, 'b']	['a', 1, 2, 'b']	concatenation; same as ['a', 1].extend([2,'b'])
*	[0] * 3	[0, 0, 0]	list repetition

Like the characters in a string, items in a list are indexed from the front with positive indices starting with 0, and from the back with negative indices starting with −1. The value of the item with index *i* is denoted *listName*[*i*]. For instance, the value of team[1] is 2014, and the value of words[−2] is "spam".

Some list functions and methods are shown in Table 2.5.

Note: After the **del** function or the **remove** method is executed, the items following the eliminated item are moved one position left in the list. After the **insert** method is executed, the items having index greater than or equal to the stated index are moved one position to the right in the list.

Example 1 **Grades** The following program requests five grades as input and displays the average after dropping the two lowest grades. The grades are placed into the list *grades*, the two lowest grades are removed from the list, and the **sum** and **len** functions are used to calculate the average of the remaining grades.

```
## Calculate average of grades.
grades = []   # Create the variable grades and assign it the empty list.
num = float(input("Enter the first grade: "))
grades.append(num)
num = float(input("Enter the second grade: "))
grades.append(num)
num = float(input("Enter the third grade: "))
grades.append(num)
num = float(input("Enter the fourth grade: "))
grades.append(num)
num = float(input("Enter the fifth grade: "))
grades.append(num)
minimumGrade = min(grades)
grades.remove(minimumGrade)
minimumGrade = min(grades)
grades.remove(minimumGrade)
average = sum(grades) / len(grades)
print("Average Grade: {0:.2f}".format(average))
```

[Run]

```
Enter the first grade: 89
Enter the second grade: 77
Enter the third grade: 82
Enter the fourth grade: 95
Enter the fifth grade: 81
Average Grade: 88.67
```

The value of the item having index *i* can be changed with a statement of the form

```
listName[i] = newValue
```

For instance, after the statement `words[1] = "eggs"` is executed, the value of *words* will be ["spam", "eggs"].

Note: In Section 2.2 we mentioned that any code enclosed in a pair of parentheses can span multiple lines. The same is true for code enclosed in a pair of square brackets. Therefore, the statement

```
team = ["Seahawks", 2014, "CenturyLink Field"]
```

can be written

```
team = ["Seahawks", 2014,
        "CenturyLink Field"]
```

Slices

A **slice** of a list is a sublist specified with colon notation. It is analogous to a slice of a string. Some slice notations are shown in Table 2.6.

TABLE 2.6 Meanings of slice notations.

Slice Notation	Meaning
list1[*m*:*n*]	list consisting of the items of *list1* having indices *m* through *n*−1
list1[:]	a new list containing the same items as *list1*
list1[*m*:]	list consisting of the items of *list1* from list1[*m*] through the end of *list1*
list1[:*m*]	list consisting of the items of *list1* from the beginning of *list1* to the element having index *m*−1

Note: The del function can be used to remove a slice from a list. Also, if the item of index *m* is not to the left of the item of index *n*, then list1[*m*:*n*] will be the empty list.

Some examples of slices are shown in Table 2.7.

TABLE 2.7 Examples of slices where list1 = ['a', 'b', 'c', 'd', 'e', 'f'].

Example	Value
list1[1:3]	['b', 'c']
list1[−4:−2]	['c', 'd']
list1[:4]	['a', 'b', 'c', 'd']
list1[4:]	['e', 'f']
list1[:]	['a', 'b', 'c', 'd', 'e', 'f']
del list1[1:3]	['a', 'd', 'e', 'f']
list1[2:len(list1)]	['c', 'd', 'e', 'f']
(list1[1:3])[1]	'c' (This expression is usually written as list1[1:3][1])
list1[3:2]	[], the list having no items; that is, the empty list

The *split* and *join* Methods

The **split** and **join methods** are extremely valuable methods that are inverses of each other. The split method turns a single string into a list of substrings and the join method turns a list of strings into a single string.

In general, if *strVar* has been assigned a string of the form "*value0*,*value1*,*value2*, . . . , *valueN*", then a statement of the form

```
L = strVar.split(",")
```

creates the list *L* containing the *N* + 1 string values as its items. That is, the first item of the list is the text preceding the first comma of strVar, the second item of the list is the text

between the first and second commas, . . . , and the last item of the list is the text following the last comma. The string consisting of the comma character is called the **separator** for the statement above. Any string can be used as a separator. (Three common separators consisting of a single-character string are ",", "\n", and " ".) If no separator is specified, the split method uses whitespace as the separator, where **whitespace** is any string whose characters are newline, vertical tab, or space characters. The split method will play a vital role in Chapter 5.

The join method, which is the inverse of the split method, converts a list of strings into a string value consisting of the elements of the list concatenated together and separated by a specified string. The general form of a statement using join and having the string "," as separator is

```
strVar = ",".join(L)
```

Example 2 **The split Method** The following statements each display the list `['a', 'b', 'c']`.

```
print("a,b,c".split(','))
print("a**b**c".split('**'))
print("a\nb\nc".split())
print("a b c".split())
```

Example 3 **The join Method** The following program shows how the join method can be used to display the items from a list of strings.

```
line = ["To", "be", "or", "not", "to", "be."]
print(" ".join(line))
krispies = ["Snap", "Crackle", "Pop"]
print(", ".join(krispies))
```

[Run]

```
To be or not to be.
Snap, Crackle, Pop
```

Text Files

Values used in a Python program reside in memory and are lost when the program terminates. However, if a program writes the values to a file on a storage device (such as a hard disk or a flash drive), any Python program can access the values at a later time. That is, files create long-term storage of data.

A **text file** is a simple file consisting of lines of text with no formatting (that is, no bold or italics) that can be created and read with Notepad (on a PC) or TextEdit (on a Mac). Usually, text files have the extension *txt*. A text file actually can be created with any word processor. For instance, after a document is created in Word, you can invoke *Save As* and then select "Save as type: Plain Text (*.txt)" to save the document as a text file. Also, any existing text file can be opened and edited in Word. Each line of a text file (except possibly the last line) ends with a newline character.

The lines of a text file (stripped of their newline characters) can be placed into a list with code of the form

```
infile = open("Data.txt", 'r')
listName = [line.rstrip() for line in infile]
infile.close()
```

The next three chapters explain how this statement carries out the task. For now, let's just assume it does the job.

If the data in a text file is all numbers, the process in the preceding paragraph produces a list consisting of strings, with each string holding a number. For a file of numbers, we can place the numbers into a list with code of the form

```
infile = open("Data.txt", 'r')
listName = [eval(line) for line in infile]
infile.close()
```

The *tuple* Object

Tuples, like lists, are ordered sequences of items. The main difference between tuples and lists are that tuples cannot be modified directly. That is, tuples have no **append**, **extend**, or **insert** methods. Also, the items of a tuple cannot be directly deleted or altered. All other **list** functions and methods apply to tuples, and its items can be accessed by indices. Tuples also can be sliced, concatenated, and repeated.

Tuples are written as comma-separated sequences enclosed in parentheses. However, they can often be written without the parentheses. For instance, the statements

`t = ('a', 'b', 'c')` and `t = 'a', 'b', 'c'`

create the tuple *t* and assign it the same value. However, **print** functions always display tuples enclosed in parentheses.

Example 4 **Tuple Functions** The following program shows that tuples have several of the same functions as lists.

```
t = 5, 7, 6, 2
print(t)
print(len(t), max(t), min(t), sum(t))
print(t[0], t[-1], t[:2])
```

[Run]

```
(5, 7, 6, 2)
4 7 2 20
5 2 (5, 7)
```

A statement such as

```
(x, y, z) = (5, 6, 7)
```

creates three variables and assigns values to them. The statement also can be written

```
x, y, z = 5, 6, 7
```

which can be thought of as making three variable assignments with a single statement.

Example 5 Swap Values The following program swaps the values of two variables. In essence, the third line of the program is assigning the tuple (6, 5) to the tuple (x, y)

```
x = 5
y = 6
x, y = y, x
print(x, y)
```

[Run]

```
6 5
```

Nested Lists

So far, all items in lists and tuples have been numbers or strings. However, items also can be lists or tuples. Lists of tuples play a prominent role in analyzing data. If *L* is a list of tuples, then L[0] is the first tuple, L[0][0] is the first item of the first tuple, L[−1] (same as L[len(L)−1]) is the last tuple, and L[−1][−1] is the last item of the last tuple. An expression such as L[0][0] can be thought of as (L[0])[0].

Example 6 U.S. Regions The list *regions* contains four tuples, with each tuple giving the name and 2010 population (in millions) of a region of the United States. The following program displays the 2010 population of the Midwest and calculates the 2010 population of the United States.

```
regions = [("Northeast", 55.3), ("Midwest", 66.9),
           ("South", 114.6), ("West", 71.9)]
print("The 2010 population of the", regions[1][0], "was", regions[1][1],
      "million.")
totalPop = regions[0][1] + regions[1][1] + regions[2][1] + regions[3][1]
print("Total 2010 population of the U.S: {0:.1f} million.".format(totalPop))
```

[Run]

```
The 2010 population of the Midwest was 66.9 million.
Total 2010 population of the U.S: 308.7 million.
```

Immutable and Mutable Objects

An **object** is an entity that holds data and has operations and/or methods that can manipulate the data. Numbers, strings, lists, and tuples are objects. When a variable is created with an assignment statement, the value on the right side becomes an object in memory, and the variable references (that is, points to) that object. When a list is altered, changes are made to the object in the list's memory location. However, when a variable whose value is a number, string, or tuple, has its value changed, Python designates a new memory location to hold the new value and the variable references that new object. We say that lists can be changed in place, but numbers, strings, and tuples cannot. Objects that can be changed in place are called **mutable**, and objects that cannot be changed in place are called **immutable**. Figure 2.29 shows eight lines of code and the memory allocations after the first four lines of code have been executed and after all eight lines have been executed.

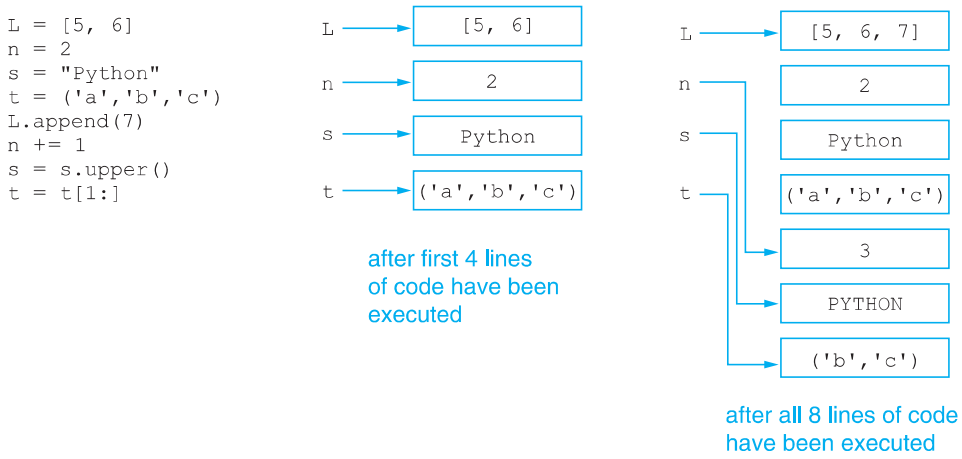

FIGURE 2.29 **Memory allocation corresponding to a program.**

Copying Lists

If the variable *var1* has a mutable value (such as a list), then a statement of the form **`var2 = var1`** results in *var2* referencing the same object as *var1*. Therefore, any change to the value of *var2* affects the value of *var1*. Consider the following four lines of code:

```
list1 = ['a', 'b']  # Lists are mutable objects.
list2 = list1       # list2 will point to the same memory location as list1
list2[1] = 'c'      # Changes the value of the second item in the list object
print(list1)
```

[Run]

```
['a', 'c']
```

In the second line of code, the variable *list2* references the same memory location as *list1*. Therefore, any changes to an item in *list2* produces the same change in the value of *list1*. This effect will not occur if the second line of code is changed to either **`list2 = list(list1)`** or **`list2 = list1[:]`**. In those cases, *list2* will point to an object in a different memory location containing the same value as *list1*. Then the third line of code will not affect the memory location pointed to by *list1* and so the output will be **`['a', 'b']`**.

Indexing, Deleting, and Slicing Out of Bounds

Python does not allow out of bounds indexing for individual items in lists and tuples, but does allow it for slices. For instance, if

```
list1 = [1, 2, 3, 4, 5]
```

then **`print(list1[7])`**, **`print(list1[-7])`**, and **`del list1[7]`** trigger the Traceback error message *IndexError*.

If the left index in a slice is too far negative, the slice will start at the beginning of the list, and if the right index is too large, the slice will go to the end of the list. For instance,

`list1[-10:10]` is **`[1, 2, 3, 4, 5]`**
`list1[-10:3]` is **`[1, 2, 3]`**

`list1[3:10]` is `[4, 5]`
`del list1[3:7]` is `[1, 2, 3]`

Comments

1. When **max** and **min** are applied to lists containing strings, lexicographical order is used to compare two strings. Lexicographical ordering of strings is discussed in Section 3.1.
2. The empty tuple is written as an empty pair of parentheses.
3. A single-item tuple must have a trailing comma, such as (0,).
4. The **list** function converts tuples or strings to lists. For instance, the value of `list(('a', 'b'))` is ['a', 'b'], and the value of `list("Python")` is ['P', 'y', 't', 'h', 'o', 'n'].
5. The **tuple** function converts lists or strings to tuples. For instance, the value of `tuple(['a', 'b'])` is ('a', 'b'), and the value of `tuple("spam")` is ('s', 'p', 'a', 'm').
6. Tuples are more efficient than lists and should be used in situations where no changes will be made to the items. They execute faster, tie up less memory, and "write-protect" data. In Chapter 5 we discuss a powerful Python object called a *dictionary*. An important feature of dictionaries requires the use of tuples.
7. As we have seen, the values of an item in a list can be altered by using its index in a statement of the form `listName[i] = newValue`. Even though the characters in a string and the items in a tuple can be accessed with indices, statements such as `stringName[i] = newValue` and `tupleName[i] = newValue` are not valid.
8. The operator += performs an augmented assignment for lists and tuples.
9. We have discussed the core objects *numbers*, *strings*, *lists*, and *tuples*. Two other important core objects, *sets* and *dictionaries*, will be introduced in Chapter 5. Although they could have been presented in Section 2.4, we decided to postpone their presentation until they are needed.

Practice Problems 2.4

1. Determine the output of the following program.

```
companies = [("Apple", "Cupertino", "CA"), ("Amazon.com", "Seattle", "WA"),
             ("Google", "Mountain View", "CA")]
(name, city, state) = (companies[1][0], companies[1][1], companies[1][2])
print(name, " is located in ", city, ", ", state, '.',sep = "")
```

2. Determine the output of the following program.

```
a = 2
b = 3
print((a + b,))
print((a + b))
print(())
```

3. Do the statements `s = 'a' + 'b'` and `s = "".join(['a', 'b'])` assign the same value to the variable *s*?

EXERCISES 2.4

In Exercises 1 through 48, assume that the list countries contains the names of fifty countries in the world, and determine the output displayed by the lines of code.

countries = ["Japan", "India", "Algeria", "Brazil", "Angola", "England", "Argentina", "Portugal", "China", "Australia", "Austria", "Ghana", "Bahamas", "Bangladesh", "Belgium", "Bhutan", "Bosnia", "Cameroon", "Canada", "Denmark"]

1. `print(countries[2], countries[-1])`
2. `print(countries[3], countries[-3])`
3. `print(countries[18], countries[16])`
4. `print(len(countries))`
5. `print(countries[-1], countries[19])`
6. `print(countries.index("Cameroon"))`
7. `print(countries.index("Ghana")`
8. `print(countries.index(countries[10]))`
9. `print(countries[len(countries) - 1], countries[-1])`
10. `print(countries[0].upper())`
11. `countries[0] = countries[0].lower()`
    `print(countries[0])`
12. `countries.insert(5, "Germany")`
    `print(countries[5])`
13. `countries.append("Nigeria")`
    `print(countries[20])`
14. `countries.insert(11,"Nepal")`
    `print(countries.index("Ghana"))`
15. `del countries[4]`
    `print(countries[4])`
16. `del countries[3]`
    `print(countries.index("Argentina"))`
17. `print(countries[2:5])`
18. `print(countries[-1:4])`
19. `print(countries[-5:-2])`
20. `print(countries[4:-1])`
21. `print(countries[:10])`
22. `print(countries[10:])`
23. `print(countries[-3:])`
24. `print(countries[:-1])`
25. `print(countries[3:3])`
26. `print(countries[-1:-4])`
27. `print(countries[1:10][2]))`
28. `print(countries[-2:len(countries)])`
29. `print(countries[:][5])`
30. `print(countries[-4][-4])`
31. `print(len(countries[10:20]))`
32. `print(len(countries[-20:]))`
33. `print(len([]))`
34. `print(len(countries[:]))`
35. `print(len(countries[1:-1]))`
36. `print(len(countries[2:-2]))`
37. `countries.extend(["Algeria","Cuba"])`
    `print(countries[-3:])`
38. `countries.append(["New Zealand", "Norway"])`
    `print(countries[-3:])`
39. `del countries[-2]`
    `countries.insert(-1, "Mangolia")`
    `print(countries[-3:])`
40. `countries[1] = "Poland"`
    `print(countries[:3])`

41.
```
del countries[1]
countries.insert(1, "Russia")
print(countries[:3])
```

42.
```
print(countries[-4].split())
states.insert(-1, "Seward's Folly")
print(states[-3:])
```

43.
```
list2 = countries[2].split() +
countries[-4].split()
list2.remove("Algeria")
print(list2)
```

44.
```
print((',').join(countries[1:4]))
```

45.
```
print(('-').
join(countries[-10:-5]))
```

46.
```
countries.remove(countries[-4])
print(countries[-4])
```

47.
```
print(('*').
join(countries[-6:-3]))
```

48.
```
countries[-1].append("Spain")
print(countries[-1])
```

In Exercises 49 through 54, assume that *list1* contains 100 items. Determine the number of items in each of the slices.

49. `list1[-8:]`

50. `list1[:8]`

51. `list1[:]`

52. `list1[-8:-1]`

53. `list1[8:8]`

54. `list1[1:-1]`

In Exercises 55 through 58, assume that the list *nums* = [6, 2, 8, 0], and determine the output displayed by the line of code.

55.
```
print("Largest Number:", max(nums))
```

56.
```
print("Length:", len(nums))
```

57.
```
print("Total:", sum(nums))
```

58.
```
print("Number lot", sum(nums) / list(nums))
```

In Exercises 59 through 94, determine the output displayed by the lines of code.

59.
```
L = ["sentence", "contains", "five", "words."]
L.insert(0, "This")
print(" ".join (L))
del L[3]
L.insert(3, "six")
L.insert(4, "different")
print(" ".join (L))
```

60.
```
L = ["one", "for", "all"]
L[0], L[-1] = L[-1], L[0]
print(L)
```

61.
```
name = input("Enter name with two parts: ")
L = name.split()
print("{0:s}, {1:s}".format(L[1], L[0]))
```
(Assume the name entered is *Charles Babbage*.)

62.
```
name = input("Enter name with three parts: ")
L = name.split()
print(L[0], L[2])
```
(Assume the name entered is *Guido van Rossum*.)

63.
```
name = input("Enter name with three parts: ")
L = name.split()
print("Middle Name:", L[1])
```
(Assume the name entered is *Guido van Rossum*.)

64.
```
list1 = ['h', 'o', 'n', 'P', 'y', 't']
list2 = list1[3:] + list1[:3]
print(("").join(list2))
```

65.
```
tuple1 = ("course", "of", "human", "events", "When", "in", "the")
tuple2 = tuple1[4:] + tuple1[:4]
print((" ".join(tuple2)))
```

66.
```
list1 = ["is", "Less", "more."]
list1[0], list1[1] = list1[1], list1[0]
print(" ".join(list1))
```

67.
```
headEditor = ["editor", "in", "chief"]
print(('-').join(headEditor))
```

68.
```
carousel = ["merry", "go", "round"]
print(('-').join(carousel))
```

69.
```
motto = ["e", "pluribus", "unum"]
print(("**").join(motto))
```

70.
```
allDay = "around-the-clock"
print(allDay.split('-'))
```

71.
```
state = "New York,NY,Empire State,Albany"
stateFacts = state.split(',')
print(stateFacts)
```

72.
```
nations = "France\nEngland\nSpain"
countries = nations.split()
print(countries)
```

73.
```
nations = "France\nEngland\nSpain\n"
countries = nations.split()
print(countries)
```

74.
```
# The three lines of Abc.txt contain a b, c, d
infile = open("Abc.txt", 'r')
alpha = [line.rstrip() for line in infile]
infile.close()
word = ("").join(alpha)
print(word)
```

75.
```
# The three lines of Dev.txt contain mer, gram, pro
infile = open("Dev.txt", 'r')
dev = [line.rstrip() for line in infile]
infile.close()
dev[0], dev[-1] = dev[-1], dev[0]
word = ("").join(dev)
print(word)
```

76.
```
# The two lines of Live.txt contain Live, let
infile = open("Live.txt", 'r')
```

```
words = [line.rstrip() for line in infile]
infile.close()
words.append(words[0].lower())
quote = (" ").join(words) + '.'
print(quote)
```

77.
```
# The three lines of Star.txt contain your, own, star.
infile = open("Star.txt", 'r')
words = [line.rstrip() for line in infile]
infile.close()
words.insert(0, "Follow")
quote = (" ").join(words)
print(quote)
```

78.
```
nums = (6, 2, 8, 0)
print("Largest Number:", max(nums))
print("Length:", len(nums))
print("Total:", sum(nums))
print("Number list:", list(nums))
```

79.
```
phoneNumber = "9876543219"
list1 = list(phoneNumber)
list1.insert(3, '-')
list1.insert(7, '-')
phoneNumber = "".join(list1)
print(phoneNumber)
```

80.
```
word = "diary"
list1 = list(word)
list1.insert(3, list1[1])
del list1[1]
word = "".join(list1)
print(word)
```

81.
```
nums = (3, 9, 6)
print(list(nums))
```

82.
```
nums = [-5, 17, 123]
print(tuple(nums))
```

83.
```
word = "etch"
L = list(word)
L[1] = "a"
print("".join(L))
```

84.
```
t = (1, 2, 3)
t = (0,) + t[1:]
print(t)
```

85.
```
list1 = ["soprano", "tenor"]
list2 = ["alto", "bass"]
list1.extend(list2)
print(list1)
```

86.
```
list1 = ["soprano", "tenor"]
list2 = ["alto", "bass"]
print(list1 + list2)
```

87.
```
list1 = ["gold"]
list2 = ["silver", "bronze"]
print(list1 + list2)
```

88.
```
list1 = ["gold"]
list2 = ["silver", "bronze"]
list1.extend(list2)
print(list1)
```

89.
```
list1 = ["mur"] * 2
print("".join(list1))
```

90.
```
list1 = [0]
print(list1 * 4)
```

91.
```
t = ("Dopey", "Sleepy", "Doc", "Grumpy", "Happy", "Sneezy", "Bashful")
print(t[4:20])
```

92.
```
ships = ["Nina", "Pinta", "Santa Maria"]
print(ships[-5:2])
```

93.
```
answer = ["Yes!", "No!", "Yes!", "No!", "Maybe."]
num = answer.index("No!")
print(num)
```

94.
```
numbers = (3, 5, 7, 7, 3)
location = numbers.index(7)
print(location)
```

In Exercises 95 through 100, identify all errors.

95.
```
threeRs = ["reading", "riting", "rithmetic"]
print(threeRs[3])
```

96.
```
word = "sea"
location = numbers.index(7)
word[1] = 'p'
print(word)
```

97.
```
list1 = [1, "two", "three", 4]
print(" ".join(list1))
```

98.
```
# Four virtues presented by Plato
virtues = ("wisdom",  "courage", "temperance", "justice")
print(virtues[4])
```

99.
```
title = ("The", "Call", "of", "the", "Wild")
title[1] = "Calm"
print(" ".join(title))
```

100.
```
words = ("Keep", "cool", "but", "don't")
words.append("freeze.")
print(words)
```

101. **Analyze a Sentence** Write a program that counts the number of words in a sentence input by the user. See Fig. 2.30.

```
Enter a sentence: Know what I mean?
Number of words: 4
```

FIGURE 2.30 Possible outcome of Exercise 101.

```
Enter a sentence: Reach for the stars.
First word: Reach
Last word: stars
```

FIGURE 2.31 Possible outcome of Exercise 102.

102. **Analyze a Sentence** Write a program that displays the first and last words of a sentence input by the user. See Fig. 2.31. Assume that the only punctuation is a period at the end of the sentence.

103. **Name** Write a program that requests a two-part name and then displays the name in the form "*lastName*, *firstName*". See Fig. 2.32.

```
Enter a 2-part name: John Doe
Revised form: Doe, John
```

FIGURE 2.32 Possible outcome of Exercise 103.

```
Enter a 3-part name: Michael Andrew Fox
Middle name: Andrew
```

FIGURE 2.33 Possible outcome of Exercise 104.

104. Name Write a program that requests a three-part name and then displays the middle name. See Fig. 2.33.

Solutions to Practice Problems 2.4

1. `Amazon.com is located in Seattle, WA.`

   The list *companies* is a list of tuples whose items can be referenced as companies[0], companies[1], and companies[2]. The program references companies[1], the tuple for Amazon. The Amazon tuple's three items are referenced as companies[1][0], companies[1][1], and companies[1][2]. ***Note:*** companies[1][0] can be thought of as (companies[1])[0], companies[1][1] can be thought of as (companies[1])[1], and companies[1][2] can be thought of as (companies[1])[2].

2.
```
(5,)
5
()
```

   The arguments of the first and third **print** functions are tuples. The comma in the first **print** function indicates that the argument is a single-element tuple, and therefore the function displays a tuple. Since the argument in the second **print** function has no comma, the set of parentheses merely encloses an expression. The third **print** statement displays the empty tuple.

3. Yes. For performance purposes, the statement using **join** is superior.

CHAPTER 2 KEY TERMS AND CONCEPTS	EXAMPLES
2.1 Numbers	
int (integer) and **float** (floating point) are numeric data types.	**int:** 3, −7, 0 **float:** 3., .025, −5.5
A **variable** is a name that points to a location in memory that holds data. The data pointed to (called the *value* of the variable) can change during the execution of the program.	`price = 19.99` `numberOfGrades = 32`
The **print function** displays values of expressions separated by spaces.	`print(32, 3., .25, price)` displays `32 3.0 0.25 19.99`.
Arithmetic operators: +, *, −, /, **, // (integer division), % (modulus).	3 + 2 = 5, 3 * 2 = 6, 3 − 2 = 1, 3 / 2 = 1.5, 3 ** 2 = 8, 7 // 2 = 3, 7 % 2 = 1, 4 ** .5 = 2
Reserved words cannot be used as variable names.	*return*, *lambda*, *while*, and *if* are reserved words.
Mathematical Functions: abs, int, round.	abs(−2) = 2, int(3.7) = 3, round(1.28, 1) = 1.3
Python is **case-sensitive**.	*price* is a different variable than *Price*.
Augmented assignments combine an operator with an assignment statement.	Suppose n = 3. After `n += 2` is executed, the value of *n* is 5.
Errors: syntax (misuse of Python language), exception (error that occurs during runtime), logic (produces unintended result)	Syntax error: print((5) [should be print(5)] Exception error: num = 5 / 0 Logic error: average = 3 + 5 / 2

CHAPTER 2 KEY TERMS AND CONCEPTS	EXAMPLES
2.2 Strings	
A **string** is a data type consisting of a sequence of characters surrounded by quotation marks.	"Hello World!", 'x', "123-45-6789"
Each character of a string is identified by its relative position from the left with a non-negative **index** starting with 0 and its relative position from the right with a negative index starting with −1.	Suppose *s* has the value "Python". The value of s[3] is 'h' and the value of s[−4] is 't'. s[10] generates an *IndexError* exception.
A **slice** of a string is a substring denoted by square brackets containing a colon and possibly numbers.	Suppose *s* has the value "Python". s[2:5] is "tho", s[−3:] is "hon", and s[:] is "Python".
String functions and methods: len, find, upper, lower, count, title, rstrip.	len("ab") is 2, "ab".find('b') is 1, "ab".upper() is "AB", "Ab".lower() is "ab", "bob".count('b') is 2, "quo vadis".title() is "Quo Vadis", and "ab ".rstrip() is "ab".
Two **string operators** are concatenation (+) and repetition (*).	"ab" + 'c' is "abc" and "ha" * 3 is "hahaha".
The **input function** displays a prompt and then assigns data entered by the user to a variable.	`name = input("Enter name: ")` `age = int(input("Enter age: "))`
A **comment** is a statement preceded with a # character that documents a program.	`rate = 5   # interest rate` `# Find average grade.`

You can break a statement within parentheses or with the use of the **line-continuation** character (\).	`print("Hello",  "World!")`
	`n = 2 +\`
	`3`

2.3 Output	

When a **horizontal tab character** (\t) or a **newline character** (\n) appears in a string, the print function displays the characters following it at the next tab stop or on the next line, respectively.	`print("spam\tand\neggs")`
	[Run]
	`spam  and`
	`eggs`

`print(val1,...,valN, sep=str1, end=str2)` displays the *N* values separated by *str1* and ending with *str2*. The arguments **sep** and **end** are optional and have default values " " and "\n".	`print(1, 2, sep='*', end="")` displays `1*2` and suppresses moving cursor to a new line. `print(1, 2)` displays `1 2` and terminates printing on the current line.

CHAPTER 2 KEY TERMS AND CONCEPTS	EXAMPLES
The **expandtabs method** controls the number of positions between horizontal tab stops. (Default is 8.)	`print("a\tbc\td".expandtabs(3))` displays `a  bc d`.

The **ljust**, **rjust**, and **center methods** control the justification of data in a field of a specified width.	`print("01234567")`
	`print("spam".center(8))`
	[Run]
	`01234567`
	`  spam`

The **format method** replaces numbered placeholders of the form {n:format specifier} in a string with comma-separated arguments of the method. Some common components of the format specifier are a number giving the width of the field in which the argument is to be displayed, a symbol giving the type of justification in the field, a comma to indicate that a number is to be displayed with thousands separators, *.r*f (where *r* is a whole number) to display a number as a **float** rounded to *r* decimal places, *d* to indicate that the argument is a number, and *s* to indicate that the argument is a string.	`s = "{0:8s}{1:>10s}"`
	`print(s.format("State", "Area"))`
	`s = "{0:8s}{1:10,d}"`
	`print(s.format("Ohio", 44830))`
	[Run]
	`State          Area`
	`Ohio        44,830`
	`print("{0:.1%}".format(.4568))`
	`print("{0:9,.2f}".format(5876.237))`
	[Run]
	`45.7%`
	`  5,876.24`

2.4 Lists, Tuples, and Files—An Introduction	
A **list** is an ordered sequence of items. Items are referred by their position (called their **index**) starting at 0 from the left end or their position (starting at −1) from their right end. **Slices** of lists are defined in much the same way as slices of strings.	L = ["spam", 35, 22.8] The item "spam" can be referenced as L[0] or L[−3]. The values of both L[0:2] and L[:−1] are ["spam", 35].
List functions: del, len, max, min, sum. (The sum function applies only to lists of numbers.)	See Table 2.5.
List methods: append, clear, count, extend, index, insert, remove.	See Table 2.5.

CHAPTER 2 KEY TERMS AND CONCEPTS	EXAMPLES

A **tuple** is a sequence similar to a list except that it cannot be altered in place. The list functions mentioned above (except for del) also apply to tuples. Of the list methods mentioned above, tuples support only count and index. Tuples support **concatenation** and **repetition**.	`t = (2, 3, 1, 3)`
	`print(t[1], t.index(3), end=" ")`
	`print(t.count(3), len(t), sum(t))`
	`print(t + (7, 5))`
	`print(t * 2)`
	[Run]
	`3 1 2 4 9`
	`(2, 3, 1, 3, 7, 5)`
	`(2, 3, 1, 3, 2, 3, 1, 3)`

The **split method** converts a string containing one or more instances of a separator (usually a comma or a blank space) to a list. The **join method** concatenates a list or tuple of strings into a single string with a specified separator inserted between each pair of items.	`print("spam,eggs".split(','))`
	`print(", ".join(['spam','eggs']))`
	[Run]
	`['spam', 'eggs']`
	`spam, eggs`

The **open function** can be used to fill a list with each line of a file as an item of the list.	`infile = open("fileName", 'r')`
	`L = [line.rstrip() for line in infile]`
	`infile.close()` creates a list of strings consisting of the lines from the file.

When a list's items are all lists, the configuration is referred to as **nested lists**.	`L = [["Bonds", 762],["Aaron", 755]]` `L[0][1]` has the value `762`.
An object whose data cannot be modified in place is said to be **immutable**.	Numbers, strings, and tuples are immutable. Lists are mutable.
Referencing an item of a list or tuple with an improper index generates an ***IndexError*** Traceback message	(5, 3, 2)[6] generates an *IndexError* Traceback message.
The **list** and **tuple functions** convert tuples to lists and vice versa.	`list(2, 3)` has value `[2, 3]`. `tuple[2, 3]` has value `(2, 3)`.

CHAPTER 2 PROGRAMMING PROJECTS

1. **Make Change** Write a program to make change for an amount of money from 0 through 99 cents input by the user. The output of the program should show the number of coins from each denomination used to make the change. See Fig. 2.34.

```
Enter amount of change: 93
Quarters: 3   Dimes: 1
Nickels: 1    Cents: 3
```

FIGURE 2.34 Possible outcome of Programming Project 1.

```
Enter amount of loan: 12000
Enter interest rate (%): 6.4
Enter number of years: 5
Monthly payment: $234.23
```

FIGURE 2.35 Possible outcome of Programming Project 2.

2. Car Loan If A dollars is borrowed at r% interest compounded monthly to purchase a car with monthly payments for n years, then the monthly payment is given by the formula

$$\text{monthly payment} = \frac{i}{1 - (1 + i)^{-12n}} \cdot A$$

where $i = \frac{r}{1200}$. Write a program that calculates the monthly payment after the user gives the amount of the loan, the interest rate, and the number of years. See Fig. 2.35.

3. Bond Yield One measure of a bond's performance is its *Yield To Maturity* (YTM). YTM values for government bonds are complex to calculate and are published in tables. However, they can be approximated with the simple formula $\text{YTM} = \frac{intr + a}{b}$, where $intr$ is the interest earned per year, $a = \frac{\text{face value} - \text{current market price}}{\text{years until maturity}}$, and $b = \frac{\text{face value} + \text{current market price}}{2}$. For instance, suppose a bond has a face value of \$1,000, a coupon interest rate of 4%, matures in 15 years, and currently sells for \$1,180. Then $intr = .04 \cdot 1{,}000 = 40$, $a = \frac{1000 - 1180}{15} = -12$, $b = \frac{1000 + 1180}{2} = 1090$, and $\text{YTM} = \frac{40 - 12}{1090} \approx 2.57\%$. ***Note:*** The *face value* of the bond is the amount it will be redeemed for when it matures, and the *coupon interest rate* is the interest rate stated on the bond. If a bond is purchased when it is first issued, then the YTM is the same as the coupon interest rate. Write a program that requests the face value, coupon interest rate, current market price, and years until maturity for a bond, and then calculates the bond's YTM. See Fig. 2.36.

```
Enter face value of bond: 1000
Enter coupon interest rate: .04
Enter current market price: 1180
Enter years until maturity: 15
Approximate YTM: 2.57%
```

FIGURE 2.36 Possible outcome of Programming Project 3.

```
Enter price of item: 25.50
Enter weight of item in
pounds and ounces separately.
Enter pounds: 1
Enter ounces: 9
Price per ounce: $1.02
```

FIGURE 2.37 Possible outcome of Programming Project 4.

4. Unit Price Write a program that requests the price and weight of an item in pounds and ounces, and then determines the price per ounce. See Fig. 2.37.

5. Stock Portfolio An investor's stock portfolio consists of four Exchange Traded Funds (SPY, QQQ, EEM, and VXX). Write a program that requests the amount

invested in each fund as input and then displays the total amount invested and each fund's percentage of the total amount invested. See Fig. 2.38.

```
Enter amount invested in SPY: 876543.21
Enter amount invested in QQQ: 234567.89
Enter amount invested in EEM: 345678.90
Enter amount invested in VXX: 123456.78

ETF     PERCENTAGE
------------------
SPY        55.47%
QQQ        14.84%
EEM        21.87%
VXX         7.81%

TOTAL AMOUNT INVESTED: $1,580,246.78
```

FIGURE 2.38 Possible outcome of Programming Project 5.

```
Enter number of miles: 5
Enter number of yards: 20
Enter number of feet: 2
Enter number of inches: 4
Metric length:
  8 kilometers
  65 meters
  73.5 centimeters
```

FIGURE 2.39 Possible outcome of Programming Project 6.

6. Length Conversion Write a program to convert a U.S. Customary System length in miles, yards, feet, and inches to a Metric System length in kilometers, meters, and centimeters. A sample run is shown in Fig. 2.39. After the numbers of miles, yards, feet, and inches are entered, the length should be converted entirely to inches and then divided by 39.37 to obtain the value in meters. The **int** function should be used to break the total number of meters into a whole number of kilometers and meters. The number of centimeters should be displayed to one decimal place. The needed formulas are as follows:

$$\text{total inches} = 63{,}360 * \text{miles} + 36 * \text{yards} + 12 * \text{feet} + \text{inches}$$
$$\text{total meters} = \text{total inches}/39.37$$
$$\text{kilometers} = \text{int}(\text{meters}/1000)$$